Project #: C-43-602  
Contract#: RESEARCH AGREEMENT DTD 2/14/91 Mod #: ADMIN REVISION  
Prime #:  
Subprojects ? : N  
Main project #:  
Project unit: SRC  
Project director(s): APPELBE W F  
Sponsor/division names: EQUIFAX  
Sponsor/division codes: 254  
Award period: 910101 to 910813 (performance) 910827 (reports)  
Sponsor amount  
Contract value  
New this change 0.00  
Total to date 23,471.49  
Funded 0.00  
Cost sharing amount  
Does subcontracting plan apply ?: N  
Title: GEMINI: A FEASIBILITY STUDY & BENCHMARK OF PARALLEL DB SYSTEMS FOR EQUIFAX ...  

PROJECT ADMINISTRATION DATA

OCA contact: Don S. Hasty 894-4820  
Sponsor technical contact  
DR. ROY SCRUGGS, MD 41-4C  
(404)740-0405  
EQUIFAX, INCORPORATED  
RESEARCH AND DEVELOPMENT  
WINDWARD EQUIFAX TECHNOLOGY CENTER  
P.O. BOX 740006  
ATLANTA, GA 30374-0006  
Security class (U,C,S,TS) : U  
Defense priority rating : N/A  
Equipment title vests with: Sponsor X  
NONE PROPOSED  
Administrative comments - 
ADMIN REVISION TO REALLOCATE BUDGETED FUNDS PER REQUEST OF PROJECT DIRECTOR.
GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 02/13/92

Project No. C-43-602__________ Center No. 10/24-6-R7137-0A0_

Project Director APPELBE W F__________ School/Lab SRC__________

Sponsor EQUIFAX/ATLANTA, GEORGIA__________________________________

Contract/Grant No. RESEARCH AGREEMENT DTD 2/14/91 Contract Entity GTRC

Prime Contract No. ________________________________

Title GEMINI: A FEASIBILITY STUDY & BENCHMARK OF PARALLEL DB SYSTEMS FOR EQUIFAX

Effective Completion Date 910813 (Performance) 910827 (Reports)

<table>
<thead>
<tr>
<th>Closeout Actions Required</th>
<th>Y/N</th>
<th>Date Submitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final Invoice or Copy of Final Invoice</td>
<td>Y</td>
<td>______</td>
</tr>
<tr>
<td>Final Report of Inventions and/or Subcontracts</td>
<td>N</td>
<td>______</td>
</tr>
<tr>
<td>Government Property Inventory &amp; Related Certificate</td>
<td>N</td>
<td>______</td>
</tr>
<tr>
<td>Classified Material Certificate</td>
<td>N</td>
<td>______</td>
</tr>
<tr>
<td>Release and Assignment</td>
<td>N</td>
<td>______</td>
</tr>
<tr>
<td>Other</td>
<td>N</td>
<td>______</td>
</tr>
</tbody>
</table>

Comments: _______________________________________________________________________________

Subproject Under Main Project No. ________________

Continues Project No. ________________

Distribution Required:

Project Director: Y
Administrative Network Representative: Y
GTRI Accounting/Grants and Contracts: Y
Procurement/Supply Services: Y
Research Property Management: Y
Research Security Services: N
Reports Coordinator (OCA): Y
GTRC: Y
Project File: Y
Other: N

Distribution Required: Other: N
1. Introduction

The Gemini project consisted of six tasks: an Initial Evaluation, a Parallel Database Prototype, a Prototype Benchmark, an Implementation Analysis and a Final Report. This paper summarizes the results of the project tasks and includes recommendations for future work.

1.1. Initial Evaluation

A major milestone of this task was the development of functional specifications for both ACRO and CLUE, after considerable interaction with Equifax staff (both functional specs. were sent to Equifax earlier). The end result was a model for ACRO and CLUE inquiries, which can be summarized as follows:

```plaintext
while inquiries do
  1. parse inquiry
  2. determine the set of superdescriptors $s_i$ from the inquiry
  3. candidate_set of records matching inquiry ← NULL
  4. for each superdescriptor $s_i$ do
     5. database_keys ← read index_table.$i[s_i]$
     6. add database_record_keys to candidate_set
     7. remove duplicates from candidate_set
  8. for each database_key in candidate_set do
     9. database_record ← read database[database_key]
    10. score_value ← score(database_record, inquiry)
    11. return database_records which meet score_value cutoff
    12. write inquiry log for database[database_key]
```

Sequential ACRO/CLUE Model Code

Description:

1. Parsing the input requires checking the validity of input fields, converting street addresses to a standard form, converting abbreviated names to a standard full-form (e.g., Bill → William). This need not be done on the high-performance parallel database (backend). In fact, doing it on the backend wastes cycles and a valuable resource.

2. Superdescriptors are formed from inquiry fields and are the fundamental keys used to lookup database records. Examples of inquiry fields used to lookup database records are: name(s), SSN’s, tradeline’s, address’s. In each case, the superdescriptor is formed by mapping a set of fields into a superdescriptor key that is intended to be both concise and unique (e.g., for names the standard full-form first name concatenate with the surname).
The candidate set is a list of all database records with any of superdescriptors. For example a name with a correct SSN might return 2 database record keys (identical), whereas a name with an incorrect SSN might return 2 different record keys.

Index tables are needed as it is necessary to rapidly lookup database records using more than one key. Index tables map superdescriptor values to their associated database key. Index tables may be stored in many ways (e.g., B-Trees (ACRO) or hash tables), but hash tables are best for parallel systems.

Index tables are too large to be kept in memory for even small databases. For example, if ACRO has 500M records, a SSN index table consisting of (SSN,record-id) pairs would require at least 8x500M bytes = 4 gigabytes.

The purpose of removing duplicates is that if the inquiry is correct all the superdescriptors will refer to the same record (assuming only one record matched the query).

One superdescriptor, the name superdescriptor in ACRO/CLUE, is referred to as the primary superdescriptor. It is the superdescriptor used to organize the database, and the one which is usually supplied and correct.

One possible optimization is to store all superdescriptor’s in each database record. Instead of looking up the superdescriptors first in an index table the primary superdescriptor is used to find the record, then the superdescriptors are checked against those in the database record. If any of these are correct then no index table lookup for the correct ones. This requires more database storage (maybe 10%) but would speedup inquiries if more than 90% of all inquiries have correct data. Obviously more study on an instrumented version of the prototype code needs to be done.

The scoring routine is table driven, and roughly modeled on the table driven scoring of CLUE. It is far easier to maintain than ACRO’s current hard coded scoring decisions, can be parallelized for a SIMD machine, and can easily be instrumented to gather statistical data on scoring (e.g., percentage of records which pass/fail scoring tests) that would very very useful in tuning ACRO/CLUE.

A score value is calculated for each database record in the candidate set. Based upon this, the highest scoring N records could be returned, records scoring above a given threshold, etc. Given the score, deciding which records to return is a simple ACRO or CLUE dependent function.

ACRO writes back into the database records returned by the inquiry: incrementing the count of the number of credit inquiries.

One possible optimization is to merely log these, then process these when the system load is low (e.g., daily, in batch, at night). This would eliminate the write time during inquiries.

The above model captures not just ACRO and CLUE inquiries, but almost any other online inquiry in which the input inquiry contains multiple keys (referred to as superdescriptors, which are built from inquiry fields e.g., concatenating a person’s first initial and name), and all records having these keys are retrieved and scored, then those records that meet the scoring cutoff are returned (e.g., those four with the highest score, or all those over a score threshold) as a result of the inquiry (and the output formatted). In addition, in the case of ACRO, database records are updated with a log that the record was accessed (credit inquiry).

The above does not preclude online updates to records. In fact, these could be easily added. Step 12 would be modified to write an updated database record with more changes than merely an incremented inquiry count. A major issue is database locking, especially in the presence of online updates. In the model developed below, each database record has a unique processor responsible for writing and reading that record, and hence it is not possible for two processors to simultaneously attempt to write the same record. The scheme we propose is that
when a processor requests a read of a disc block from another processor, it later sends a reply to that processor (either writing back updated data or notifying the processor that the block will not be updated). Between the time that the block is read and a reply is received the block is marked as unavailable, and other transactions attempting to access this block will be blocked. It should be noted that these conflicts will be very very rare.

Backups of the database can be done most easily while no other transactions are taking place, just as is done currently with ACRO/CLUE.

1.2. Parallel DataBase Model

In parallelizing the CLUE/ACRO model we faced several decisions, which can be summarized as follows:

0. How to parallelize the model?
1. How to distribute the data?
2. How to distribute the computation?

For each inquiry, it is clear from the above model that it is difficult to parallelize and individual inquiry. The reading of database keys can be done in parallel (but there are only a handful of keys, e.g., name, SSN, address, ...) for a given inquiry; and also the reading and scoring of database records for these keys can be done in parallel (but again there are only a handful). For a given inquiry, and ACRO/CLUE, the average maximum theoretical speedup (assuming no overhead for parallelism) for a single inquiry is at most 5 or 6.

By contrast, processing many inquiries in parallel offers significant speedup. As argued below, the speedup is within a constant factor of linear (assuming but not unreasonable sufficient I/O and interprocessor bandwidth), which implies that doubling the parallel computer configuration (number of processors and I/O channels) would roughly double the number of inquiries per second that could be processed.

There are several principles of parallelization that can be applied to determine the optimum way to parallelize ACRO/CLUE:

Static Data Partitioning
In order to effectively parallelize ACRO/CLUE it is necessary to minimize both interprocessor communication and transfer of data. Data must be statically partitioned (i.e., the database record for a given name such as Appelbe ought always to be on the same disc); the overhead for dynamic lookup of data locations is high and unnecessary.

Under this assumption, the main database records are statically allocated based upon the primary superdescriptor. For example, if the primary superdescriptor's are in the range 1 to 1,000, and there are 10 processors, then records 1-100 would be on processor 1, records 101-200 would be on processor 2, etc. This implies that superdescriptor distribution must be uniform, so that all processors are roughly equally loaded.

No replicated data
Given the size of the database, this is infeasible.

Other principles are applicable equally well to parallel or serial versions of ACRO/CLUE:

Minimize disc accesses
This implies not using indirect tables to locate disc records.

The parallel version of the ACRO/CLUE model code is based upon breaking up the index files and the main database statically into equal-sized chunks for each processor. We also assume that each processor has its own local disc, upon which its chunks of the database reside. This assumption allows for discs shared among processors, but since ACRO/CLUE is highly I/O intensive, with the majority of time spent in I/O operations, significant speedup will be obtained only if the I/O bandwidth is proportional to the number of processors (for an I/O bounded system, increasing the number of processors will not speedup the application at all; only by increasing the I/O in proportion to the increase in processors will the increase in processors be effective). This implies either per-processor discs, or a very-high throughput multiplexed disc (which appears to be a local disc). A high performance computer in which only a small proportion of processors are attached to discs will have its ACRO/CLUE performance very limited.
Each superdescriptor \( s_i \) consists of two parts: \( s_i.\text{PROC} \) the processor it resides on, and \( s_i.\text{BLOCK} \) the disc block on that processor's disc. The main database is allocated as the primary superdescriptor, and the processor the has the inquiry's primary superdescriptor is responsible for doing all the inquiry processing (referred to as the primary processor, although processors holding the index table for other superdescriptors and disc blocks will act as slaves for the primary processor.

The code below works for a distributed memory MIMD system such as the nCube and can be adapted for an SIMD architecture such as the MasPar, though on the MasPar all operations would be synchronous. We assume two non-blocking message passing primitives:

- \( \text{SEND}(p,m,d) \) - send processor \( p \) the message type \( m \) with data \( d \)
- \( \text{WAIT}(p,m,d) \) - wait for the message type \( m \) from processor \( p \) with data \( d \)

The code is as follows as follows:

```
FRONT-END inquiry preprocessing:
1   parse inquiry \( I \), with \( N \) superdescriptors
2   determine the set of superdescriptors \( s_i \), \( i = 1 \ldots N \) from the inquiry
3   \( \text{SEND}(s_1.\text{PROC}, \text{PROCESS_INQUIRY}, I) \)
4   \text{for} \( i = 2 \text{ to } N \) \text{do}
5     \text{if } s_i.\text{PROC} \neq s_1.\text{PROC} \text{ then } \text{SEND} \( (s_i.\text{PROC}, \text{SECONDARY_SUPERDESCRIPTOR_REQUEST}, (s_1.\text{PROC}, s_i.\text{BLOCK})) \)

PROCESSOR inquiry processing:
6   \text{do} \text{forever}
7     \text{WAIT}(p,m,d)
8     \text{case } m \text{ do}
9      \text{PROCESS_INQUIRY:}
10     \text{candidate_set of records matching inquiry } \leftarrow \text{read index_table}_1[s_1.\text{BLOCK}]
11    \text{for } i = 2 \text{ to } N \text{ do}
12       \text{if } s_i.\text{PROC} \neq s_1.\text{PROC} \text{ then } \text{WAIT} \( (s_i.\text{PROC}, \text{SECONDARY_SUPERDESCRIPTOR_REPLY}, \text{key}_i) \)
13         \text{else}
14           \text{key}_i \leftarrow \text{read index_table}_i[s_i.\text{BLOCK}]
15           \text{add key}_i \text{ to candidate_set}
16     \text{for each } \text{database_key in candidate_set do}
17       \text{if } \text{database_key.PROC} = \text{MY_PROC_ID} \text{ then }
18          \text{database_record } \leftarrow \text{read database[database_key.BLOCK]}
19       \text{else}
20          \text{SEND} \( (\text{database_key}.\text{PROC}, \text{READ_BLOCK_REQUEST}, \text{database_key}) \)
21          \text{WAIT} \( (\text{database_key}.\text{PROC}, \text{READ_BLOCK_REPLY}, \text{data_base记录}) \)
22          \text{score_value } \leftarrow \text{score(database_record, inquiry)}
23          \text{return database_records which meet score_value cutoff}
24          \text{write inquiry log for database[database_key]}
25    \text{SECONDARY_SUPERDESCRIPTOR_REQUEST:}
26       \text{SEND} \( (s_1.\text{PROC}, \text{SECONDARY_SUPERDESCRIPTOR_REQUEST}, \text{read index_table}_1[s_1.\text{BLOCK}) \)
27    \text{READ_BLOCK_REQUEST:}
28       \text{SEND} \( (s_1.\text{PROC}, \text{READ_BLOCK_REPLY}, \text{read database[database_key.BLOCK]}) \)
29  \text{end forever}
```
Parallel ACRO/CLUE Model Code

Description:

1-5 All the preprocessing of inquiries, including routing of inquiring messages to processors, is assumed to be done by the front-end that manages operator interaction, etc. The inquiry messages generated can then be sent to the processors over a high-speed communication link or spooled to a dual-ported disc. All high-performance computers have a 'front-end' processor that could handle the preprocessing. The preprocessing requires NO I/O, and should only take less than a millisecond (allowing for a back-end rate of > 1,000 tps). Since no I/O is required for preprocessing it could also be done by the networked operator workstations. If the secondary superdescriptor is on the same processor as the primary descriptor then no message send (5) is needed or sent.

6-7 Every processor waits upon incoming messages, either PROCESS_INQUIRY, in which case the processor is the inquiry 'master', or a REQUEST to read either a local superdescriptor index table or local database chunk.

8-22 The main PROCESS_INQUIRY code is almost exactly the same as the sequential code except that the secondary superdescriptor keys and database records may lie on other processors. In the case of superdescriptors a request to fetch the index key will have been sent in the preprocessing stage. In the case of a block request for a min database record the process inquiry code sends out the request. In both cases the process inquiry code waits for a reply. There is the possibility of a deadlock in the above code, if simultaneously two inquiries are sent to different primary processors, each of which sends requests to the other for disc blocks. This can be eliminated by going to the outer level request loop after sending a request.

If non-local discs are directly accessible, then it may be possible to bypass the READ_BLOCK_REQUEST and access the remote disc directly. However, if each disc block can be read and written by more than one processor the strategy for managing locking blocks is potentially complex.

By allocating the main database records so that the database records for any primary superdescriptor are on the same processor as its superdescriptor index table remote disc accesses are minimized. For example, assume an inquiry with all superdescriptors (e.g., name, address, SSN) correct and a single database record for the inquiry. All the index tables will refer to the same record, which will be the same as the primary superdescriptor (name, say). In this common case no remote disc reads for database blocks will be needed.

23 The write of the inquiry log may require remote disc accesses, if the main database records are updated and the database record was fetched from a remote disc. This code has been omitted but is fairly obvious.

24-27 Requests always result in a read from disc and a reply to the primary processor with the data read.

1.3. Parallel Database Prototype

The parallel database code is operational on the nCube, and will be ported to the MasPar by early June. The code itself is highly modular. Only a small percentage is ACRO/CLUE or target specific.

1.4. Prototype Benchmark

The majority of the effort in this project has been devoted to understanding ACRO/CLUE, documenting them, developing a sequential prototype, generating a testdata set, then navigating the variegated features of the nCube. The benchmarks below are hence preliminary. We have fairly carefully optimized the design of the parallel algorithm, and have also optimized disc accesses.

The benchmarks below are based upon a database of 10,000 records and 200 inquiries with 1 to 5 descriptors.

All times are averaged over 200 iterations of the inquiry processing loop for the nCube:
1 node, 1 disk
  test 1: 17.799 seconds   11.24 TPS
  test 2: 17.666 seconds   11.32 TPS

2 nodes, 2 disks
  test 1: 12.683 seconds   15.77 TPS
  test 2: 12.533 seconds   15.96 TPS

4 nodes, 4 disks
  test 1: 6.133 seconds    32.61 TPS
  test 2: 6.266 seconds    31.92 TPS

We are working on further timings for the other configurations (8x4, 8x8, and 16x8 processors/discs). Based upon the I/O to CPU ratios the number of discs bounds performance (with N processors, N discs are needed for effective speedup). The discs used are 1 gigabyte. Extrapolating the above, 200TPS could be easily exceeded with about a 32 node, 32 disc configuration.

For the SUN 3, 64 TPS were achieved for ACRO/CLUE. This is faster than the nCUBE 4 node, 4 disc primarily because the SUN version has no parallel processing overhead, and a faster, smaller, local disc (not networked as in the nCube). However, unlike the nCUBE, the SUN 3 version cannot be speeded up significantly (by either a faster disc or processor), whereas the nCUBE in a 64 node, 64 disc configuration would reach close to 400TPS.

1.5. Parallel Database Model Evaluation

The model evaluation is preliminary as the final coded version of the parallel ACRO/CLUE model is not complete, but will be finalized by early June.

The sequential time to process an inquiry with N superdescriptors, that matches M database records is:

\[ T_{\text{serial}} = T_{\text{disc\_read}} \times (M + N) + T_{\text{processing}} \]

where \( T_{\text{disc\_read}} \) is the mean disc read time and \( T_{\text{processing}} \) is the mean inquiry CPU processing time. For the SUN 3 typical numbers for these are

\[ (M + N)T_{\text{disc\_read}} = 0.0194 \text{sec.} \]
\[ T_{\text{processing}} = 0.00958 \text{sec.} \]

The important point to note is that: Disc I/O time is about double the CPU time, per transaction. ACRO/CLUE is totally I/O bound and changes in disc performance would not affect this significantly.

For the parallel implementation, calculating the expected time to process an inquiry depends upon the distribution of inquiries to processors. Clearly, a uniform distribution yields the best performance, a poisson distribution is likely, and a skewed distribution in which one processor received all the inquiry transactions would lead to very poor performance. Fortunately, with a good superdescriptor hash function and a large number of processors (>100) the probability of a skewed distribution temporarily occurring is very small. For a single transaction, assuming that all processors cooperating in processing the transaction are not processing other transactions, the time to process an inquiry with N superdescriptors, that matches M database records on different processors is:

\[ T_{\text{parallel}} = T_{\text{WAIT}} + T_{\text{disc\_read}} + T_{\text{SEND}} + (N - 1) \times T_{\text{WAIT}} + (M - 1) \times T_{\text{SEND}} + T_{\text{WAIT}} + T_{\text{READ}} + (M - 1) \times T_{\text{WAIT}} + T_{\text{processing}} \]

If the time for interprocessor communication is less than about 0.5 times the time for a disc operation then \( T_{\text{parallel}} \) is less than \( T_{\text{serial}} \) even for a single inquiry. The reason for this is that the index and database reads \( (M + N) \) are performed in parallel in \( T_{\text{parallel}} \), which has an overhead of about \( N + M \) messages. If the processors involved in processing an inquiry do not conflict with those processing another query then there is linear speedup. A fairly quick analysis shows that \( P \) processors can process about \( \frac{P}{PQ} \) inquiries in parallel without
seriously conflict, where $PQ < M + N$ is the number of processors involved in an inquiry.

Further analysis will be possible after the parallel prototype benchmarks are complete.

More studies need to be done on real ACRO and CLUE inquiry streams. Ideally, we would like to work with Equifax Personnel to ‘capture’ a fairly long log of typical ACRO and CLUE inquiries, and also capture a part of the database. Currently we have used real names and addresses, but have generated all the other data. We also have not received any statistical data about the number of superdescriptors used on average; the probability that inquiry data will be in error (leading to reads of remote blocks for scoring, etc.)

1.6. System Implementation Analysis

The current CLUE/ACRO prototype code is approximately 1,600 lines of C code. To implement a full-fledged C version of either ACRO or CLUE inquiry processing we estimate would increase code size to 2,700 lines of C. Missing from ACRO at present are some input preprocessing, gender table lookup (which maps a name to a gender), state table lookup (which maps a 2 character state abbreviation to a number) and a few of the scoring tests (14 are currently implemented). A full-fledged version of CLUE would differ from the CLUE version by approximately 500 lines, primarily in the scoring routines, a different set of superdescriptors, and different input scanning. This is considerably more compact than the current ACRO or CLUE implementations, because of the use of C and UNIX system calls. Missing features, that would need to be added for a full-fledged

By full-fledged we mean a version of ACRO and CLUE with all the functionality of either systems inquiries (i.e., supports at least the same set of inquiries and responses) and retrieves a comparable set of database records. By comparable, we mean that in the vast majority of cases the database records retrieved would be identical, though in the rare cases where they differed an operator would not be able to detect an error (i.e., ambiguous queries). Making a new version of ACRO or CLUE 100% compatible with the existing version seems unnecessary, and would require in the limit incalculable resources (generate EVERY POSSIBLE query for both databases). We claim that our version of ACRO/CLUE, with its table-driven scoring, is much easier to tune, adapt, and monitor to improve precision and recall (to ensure that operator queries result in nothing but successful results returned). Monitoring is critical to determine, for example, what percentage of inquires have the wrong sex, or wrong SSN, etc. Monitoring is only possible if its overhead is tolerable (say 20%). In both ACRO and CLUES’ current incarnations, as far as we are aware, there is almost no continual monitoring to improve the databases performance (precision and recall) by tuning the scoring routines. This is partly due to the complexity of the current code and the nearly 100% utilization of the systems (so that no ‘cycles’ are available for monitoring). Improved performance would result in both decreased operator dissatisfaction and errors in the database and subsequent litigation and costly manual database updates. It cannot be emphasized enough that an important spinoff of recoding ACRO/CLUE so that monitoring was possible.

Our experience so far with parallel systems, including the nCube and MasPar, is that although they excel at computation, and have potentially high I/O throughput, they have very limited software support. For example, on the nCube:

- The file system is extremely fragile (a simple error led to the entire disc appearing to be full even when files were opened read-only!)
- The programmers manual is short on information and in some places inconsistent
- nCube’s file-I/O is not done through standard UNIX library functions - extensive rewrites were required
- The nCube C compiler is not ANSI-C compliant
- nCube uses a different little-endian byte ordering; ‘swap’ must be explicitly called when transferring data to and from the host

Unfortunately, such problems are endemic to the high-performance computer industry.

Although different vendors of high-performance computers have different strengths and weaknesses, none of them have anything approaching the software support, libraries, tools, or applications VARS, that mainframes have. Unfortunately, this situation is unlikely to change in the near future. If EQUIFAX wants to be on the ‘cutting edge’, it must by necessity be there without much software support. Nevertheless, high performance
computers do offer dramatic potential cost/performance benefits provided that these benefits can be safely realized. The solution is threefold:

- Only use high-performance computers for what they do best: parallel computation and I/O. All inquiry preprocessing and postprocessing (e.g., validity checking of input fields, construction of superdescriptors, prompting operators, formatting output) should be done by front-end processors (from PC's to mainframes, if input preprocessing and postprocessing was a portable C program a central CICS mainframe could be avoided).
  Ideally, the high-performance computer would continually read transactions that were fed to it from a front-end either through a high-speed channel or a spooled dual ported disc.

- Write applications in portable code, such as C, that can be moved to another high-performance computer readily.

- Ensure that when a high-performance computer goes ‘online’ that it has been heavily tested and is well supported.

It is too early to make recommendations, except to say that our expectations that ACRO/CLUE could be ported successfully to high-performance computers with major performance gains have been vindicated despite many problems we encountered in ‘understanding’ ACRO and CLUE, creating a general purpose model, implementing it on both sequential and quirky parallel machines.

1.7. CONCLUSIONS

The major conclusions from GEMINI are highly encouraging:

- ACRO and CLUE share the same structure, and one common core can implement both applications
- Using modern techniques, ACRO/CLUE can be implemented portably in C/UNIX, with effectively the same functionality as the production applications, in only a few thousand lines of code
- On a relatively small nCUBE, we could achieve performance only a small factor away from production ACRO (or CLUE). A 64 node, 64 disc configuration would exceed ACRO’s performance, and contain all its data.
- Disc I/O time is about double the CPU time, per transaction. ACRO/CLUE is I/O bound by about a factor 2 and changes in disc performance or disc caches or enhancements to the scoring routines would not affect this significantly. Correspondingly, faster processors are unnecessary.