Use of the Hypercube for Symbolic Quantum Chromodynamics†

A. Kolawa, G. C. Fox

Caltech Concurrent Computation Program
Mail Code 206-49
California Institute of Technology
Pasadena, CA 91125

March 1988

ABSTRACT

A new numerical approach by Furmanski and Kolawa to quantum chromodynamics is based on diagonalizing the underlying Hamiltonian. This method involves the generation of states by repeated action of a potential operator. This symbolic calculation is dominated by the time it takes to search the database of existing states to verify if a generated state is identical to one previously found. We implement this algorithm on the Caltech/JPL Mark II hypercube and analyze its performance of both a simple database search and one optimized for this application. We show that the hypercube performance can be modelled in a fashion similar to conventional numerical (loosely synchronous) applications.

† Work supported in part by DOE grant DE-FG03-85ER25009, the Program Manager of the Joint Tactical Fusion Office, and the ESD division of the USAF, as well as grants from IBM, and SANDIA.

1. Introduction

We have used the hypercube at Caltech extensively for the numerical Monte Carlo approach to Lattice gauge theories [Otto 84], [Flower 87], [Apostolakis 88]; the most popular approach to solving the fundamental equations of particle physics. In this paper, we would like to illustrate the flexibility of the hypercube by discussing its use in a fundamentally different approach to the solution of gauge theories.

Furmanski and Kolawa have developed a new approach to numerical Lattice Gauge Theory based on diagonalizing the Hamiltonian [Furmanski 86]. Such an approach is, of course, a familiar technique in chemical problems. In Sec. 2, we show that this can be implemented on a computer and that the most time consuming part involves the generation of a list of unique states. In Sec. 3, we detail the sequential algorithm showing that it can be viewed as searching and updating a database. In Sec. 4, we discuss the hashing algorithm and its extensions. In Secs. 5 and 6, we describe and analyze the concurrent implementation showing that it has good performance on the large databases associated with realistic problem parameters. Conclusions are given in the final section.

2. Description of the Physical Problem

In this paper, we will consider the solution of the SU(2) gauge field theory on a finite lattice. We will solve Schrodinger's equation where we are only interested in two lowest eigenvalues of the Hamiltonian H :

$$H|e> = E|e>. \tag{1}$$

We use the method that is fully described in [Furmanski 86] and [Kolawa 86a].

The Hamiltonian for the SU(2) gauge theory can be written as a sum of two terms. The first part D, is diagonal in the Fourier basis $|1>$ (where every link has representation I of the SU(2)). The second part of the Hamiltonian is the plaquette operator V which can change representations on links around a plaquette by 1/2. In this basis, we can look for eigenstates of Hamiltonian as:

$$|e> = \sum_{k_i} f_{k_i} |k_i> \tag{2}$$

where k counts number of plaquettes with nontrivial representations that the state can be built from (in the following k will be called the generation). i is an internal label for states with a given number of plaquettes. One can ensure that states are translationally and rotationally invariant and also orthonormal.

We can rewrite (1) as

$$(D + V)|e> = E|e>. \tag{3}$$

Multiplying (3) by the state $< 0 |$ we get:

$$<0|D|e> + <0|V|e> = <0|E|e> \tag{4}$$

or

$$f_1 <0|V|e> = E \tag{5}$$

Multipling (3) by $< 1 |$

$$Df_1 + f_0 <1|V|0> + \sum_i f_{2_i} <1|V|2_i> = Ef_1 \tag{6}$$

Similarly for state $< 2_i |$

$$Df_{2_i} + f_1 <2_i|V|1> + \sum_k f_{3_k} <2_i|V|3_k> = Ef_{2_i} \tag{7}$$

Following this procedure we can write an infinite set of linear equations which we have to truncate in order to make it solvable. This we can do by neglecting in the (n-1)'th equation the matrix element $< n\text{-}1 \, |V| \, n >$.

The computer program has to compute all the coefficients in the equations. In order to get them, we first have to produce all required states. The program which produces these states is subject of this paper. All the required states can be produced by procedure of repeated action with the operator V. First we act with V on an empty lattice (k=0) and produce one plaquette with representation 1/2 on it. This is called the first generation or a k=1 state. Then we act again on this state and we get several different states which are called the second generation with k=2. Acting again with V on the states from the second generation we produce the third generation and so on. Every time we produce a state, acting with V, we have to check if the state is on the list of states already produced. If not we must add the new state to the list. It is important to note that the states are either identical (up to a phase) or linearly independent.

3. The Algorithm as a Data Base Problem and the Sequential Implementation.

The algorithm is

1. Generate a new state
2. Compare state with current list
3. If state is distinct from those already on list, add it to list; otherwise return to step 1.

We can abstract this as a database problem by mapping the steps into:

1. Generate a database query
2. Examine database to see if query satisfied. We define success to be that state (record) already exists; failure that state is new.
3. Add new record to database if query fails; otherwise return to step 1.

The database consists of a set of records with one record per state.

The performance of the algorithm on the concurrent processor will naturally depend on such parameters as size of database; time taken to generate query; time taken to compare candidate state (query) with a single record in database. Thus the design of the concurrent algorithm must be based on the specific problem at hand. We will consider a particular example - namely the production of the third generation (k=3) of states for a $4 \times 4 \times 4$ lattice. As discussed later, the performance of our concurrent algorithm will improve as the problem size increases and so finding a good implementation for this set of parameters should work well for the larger problems of physical interest.

Table 1: Parameters of a Sample Problem
Production of Third Generation of 4^3 Lattice

Total Execution Time	1300 Seconds 6600 Seconds	(VAX11/780) (Single 8086-87 node of Caltech Hypercube)
Number of Queries (Candidate States)	2571	
Number of Distinct Records (Final number of States)	890	
Fraction of Time Spent on failed queries (new states)	0.62	
Fraction of Time Spent on successful queries (States already on list)	0.36	
Fraction of Time Spent Generating Queries	0.02	
Total number of Data base comparisons	1.06×10^6	
Average number of comparisons for successful queries	412	
Average number of comparisons for failed queries	445	

The details of this sample case are given in Table 1. In the case of successful queries (candidate state on list) the match was on the average found after searching a fraction 0.46 of entries in the database. A histogram of this fraction is given in Figure 1†. Note how flat the distribution is. As we had to search many records to find the match (and so terminate step), it will turn out to be straightforward to find a concurrent algorithm as the search time (step 2) dominates the "sequential" steps 1 and 3 corresponding to generation of query and updating database. We should also note that the comparison of a candidate state with a pre-existing state is arranged hierarchically and the comparison time varies from case to case. This inherent load imbalance due to the irregularity of the problem needs to be kept in mind for the concurrent algorithm. We would also like to note here that because of the symmetry group of the states the actual database we were searching is 48 times larger than the one presented in Table 1. We reduced the database size by keeping in memory only one representative of the symmetry group for any state and by using a special technique for fast elimination of unsuccessful queries.

4. Concurrent Implementation of Direct Search and Performance Analysis

The performance of a database search depends critically of the ability to index the records. If one has to search for an essentially arbitrary occurrence of some set of strings in a recored, then there is no practical alternative to directly searching each record. We will first use this brute force approach, even though as we describe in Sec. 5, this particular database

† Note that most of the successful queries are generated with the database nearly complete i.e. initially most queries fail, the database builds up and then most queries succeed.

can be efficiently indexed. The advantage of the simple search lies in the simplicity of the implementation and ability to compare with a performance model. We should stress that these results and performance analysis generalize to any such direct database search on a hypercube. One need only change the parameters in Equation (10) below.

We will describe the implementation on the Caltech Mark II hypercube which consists of 32 8086-8087 based nodes with the 5 dimensional hypercube geometry [Tuazon 85]. The system has a separate 8086 based controller in which the control process runs. The algorithm does not depend on the particular machine architecture and will work well on any MIMD architecture with large grain size (reasonable memory per node) and ability to send messages between processors. Suppose we have a total of N nodes in the concurrent processor (excluding the control processor).

We will decompose the problem in a simple fashion storing N equal fractions (1/N) of the database in each node. The algorithm is as follows:

1a) Control process sends to each processor the information necessary to generate the next group of states or in our more general language a group of queries. These 20-30 queries correspond to actions of the operator V on all second generation states.

Each node empties its message buffer and accepts message from control process.

1b) In each node, we loop over queries in this group.

2a) Each query is first checked to see if it is referenced in a message buffer corresponding to another processor having found it in its local database. (See steps 3a, c).

2b) If this check fails, we compare queries with records currently stored in given node. This is identical to sequential algorithm but corresponds to a database that is 1/N of the size.

3a) If the query is successful, a message is sent to all processors indicating this so that they may terminate consideration of query corresponding to this state.

3b) If the query is unsuccessful, the node either moves to the next one or stores the record corresponding to new state if a simple hashing algorithm determines that this new record (state) is to be stored in this particular node's database.

3c) At any time, messages may be received from another processor. This message can concern either the current query, a query already considered or a future query. In the first two cases, appropriate action is taken - which might require deletion from database of a past state whose query failed in current node but which was later found in database stored in another node. If message concerns a future state, it is stored in a buffer to be used in step 2a).

3d) A message is sent to control process when the node finishes the given group of queries. When the control process receives N such completion messages it moves on to the next group.

The steps above are labeled so that, for instance, 1a and 1b of the concurrent algorithm correspond to step 1 of the sequential case in Sec. 3. Essential features of the concurrent algorithm are

- The generation of states (queries) is done in every node i.e. no concurrency is achieved in this step.
- The nodes operate asynchronously on groups of queries and so achieve approximate load balancing that averages over fluctuations in query comparison time.
- A message is sent as soon as a query succeeds in a given node and the other nodes can therefore immediately terminate any current or future comparison of this query.
- The algorithm uses in an essential fashion not only a general message passing system but also that the receipt of message interrupts the node processor. This is necessary to easily process possible termination of current query. The relevant software was built in terms of the message passing operating system JDOS built internally at Caltech [Johnson 85].

In figure 2, we plot inverse of the efficiency ϵ (sequential time divided by N times the time taken on N node concurrent implementation) as a function of N in the range 1 to 32. Even for N=32 we find an efficiency of 60% in spite of the modest size of the local database with a maximum of 28 entries. Clearly, the algorithm will perform better as the size of the database increase. The approximate formula for inverse of the efficiency is :

$$\frac{1}{\epsilon} = \frac{N \cdot \text{concurrent time}}{\text{sequential time}} = \tag{8}$$

$$= \frac{N(\frac{gMQ}{c_1 N} + dQ + c_2 bQ \log_2 N)}{\frac{gMQ}{c_1} + dQ} \tag{9}$$

Here we put c_1 equal 2 because in average only half of the database is searched for query (see figure 1, table 1). The constant c_2 we took 2/3 because nodes communicate only for failed queries. With these values of constants c_1 and c_2 equation (9) takes form:

$$\frac{1}{\epsilon} = \frac{1 + \frac{2d}{gM}N + \frac{4b}{3gM}N\log_2 N}{1 + \frac{2d}{gM}}, \tag{10}$$

where

g - time spent on query per state,

d - time spent to produce query,

b - time to broadcast message,

Q - number of queries,

M - size of the database.

The logarithmic dependence on N in Eq. (10) is expected as this is the dependence of communication time on N for the broadcast message passing used in the algorithm. Constants d and g can be calculated, using data included in the table 1, from the following formulas :

$$g = \frac{\text{sequential time}}{\text{number of database comparisons}} = 5.76ms \tag{11}$$

$$d = \frac{\text{sequential time*fraction of time spent generating queries}}{\text{number of queries}} = 51ms \tag{12}$$

Equation (10) is compared with the results in Fig. 2. Values of the constants g and d are taken from Eqs. (11) and (12) and constant b, regarded as a free parameter, is fitted to be 1.41†. The exact value of b is difficult to predict due to the asynchronous, independent operation of nodes. One would expect $b\log_2 N$ to be roughly the time taken for a broadcast message to traverse half the cube. This model would predict b to be 1 ms, which is in satisfactory agreement with the fitted value [Kolawa 86].

In [Fox 88a], [Fox 86a], and [Fox 85a] we have shown how the performance of many problems maybe summarized in the simple form:

$$1/\epsilon = 1 + \frac{\text{constant}}{n^{1/d_S}}. \tag{13}$$

Here d_S is the dimension of the problem and n is the grain size - the amount of information stored in each node. In our case $n=M/N$ and we find the form:

† The value of b could be reduced by about a factor of two using a full assembly language coding [Kolawa 86].

$$1/\epsilon = 1 + \frac{(\alpha + \beta \log_2 N)}{n}, \tag{10a}$$

where

$$\alpha = 2\frac{d}{g}, \ \beta = 4\frac{b}{3g}. \tag{14}$$

The "constant" in Eq. (13) is usually independent of N although we do find the logarithmic dependence seen in (10a) for the case of the Fast Fourier Transform. As in the database case, the log N dependence is characteristic of long distance communication. We therefore find that the direct database search fits in with our general systems analysis with the problem dimension d_S taking the value 1.

5. An Improved Search Algorithm

For the algorithm described in Secs. 3 and 4, the information describing the state on the list, which was kept in the memory, consisted of two separate parts:

1) coordinates of the center of the mass of the state (which played the role of key),
2) the information on how to construct a state on the lattice; this is the real record of the state.

The comparison of a new state with the list of states was done in two consequent steps. First it was checked whether the centers of mass of two states can be matched together. This test rejected about 90% of false comparisons. The remaining 10% were done in a direct way, which means that both states were generated on separate lattices and were checked link by link. If this comparison was successful then the states were assumed to be identical.

We now describe an improved technique using a state symmetry invariant key.

The improvement of the searching technique can be two-fold. First one can introduce another layer of keys - a state can be initially identified by the trace of its moment of inertia tensor. Then this number can be used to order states by arranging them in increasing numerical order of the trace keys. Finally the states can also be clustered in buckets within some range of trace keys.

Thus the searching algorithm should look as follows:

1) After a state has been generated, calculate the trace of its moment of inertia tensor, and its center of the mass.
2) Check the trace key against the list of buffers to find in which buffer the state should be in. We now do a standard binary search within each bucket. This is described in steps 3 and 7.
3) Take the middle state from the bucket.
4) Check the trace key of the generated state against the trace key of the state from the bucket, if the keys are the same go to 5, otherwise go to 7.
5) Compare the second keys - try to match centers of mass of the states if successful go to 6, otherwise go to 7.
6) Produce both states on separate lattices and compare link by link, if successful - stop, the states are the same, otherwise go to 7.
7) If the state is the last to be checked - go to 8, if the trace key of the generated state is smaller then the key of the checked state, take the middle state in the lower half of the bucket and go to 4, if the trace key of the generated state is larger than the key of the checked state take the middle key in the upper half of the bucket and go to 4. If the trace key of the generated state is the same as the checked state find if there is any other state with the same trace key. If there is go to 5 (the key with the same value of the trace key should be a neighboring state), if there is no go to 8.

8) Add the new state to the bucket which was checked in the order of increasing values of trace keys.

6. Concurrent Algorithm for the Indexed Data Base

We now describe and analyze the performance of the improved algorithm described in the last section. Clearly the details are application dependent - not every database has a "rotationally invariant trace of moment of inertia"! However, the issues and analysis technique should be reasonably general for any database allowing this type of indexing. In analysis of the performance of this algorithm we will use data presented in Sec. 4 from the second generation. This database is rather small, but for higher generations, we estimate the size of the database of order of 100,000 states which motivates the following algorithm. The basic assumptions of the new algorithm are the following:

- Every processor checks a different state produced from the same state in the previous generation.
- We keep in each node the entire index based on the trace of moment of inertia key. If the node memory is insufficient, the algorithm can easy be extended for a split index.
- The actual database is divided into N parts and each part is kept in a different processor.

With the above assumptions we can construct the following concurrent algorithm:

1a) The control process sends to each processor the information necessary to generate the next group of states, or in our more general language a group of queries. These 20-30 (2000 - 3000) queries correspond to actions of the operator V on all second (third) generation states.
Each node empties its message buffer and accepts message from control process.

1b) Each node acts with operator V at a different place on the same state and produces a different query.

2a) Each node checks the trace key of the produced state against the whole list of keys using the same logarithmic search described for the sequential algorithm.

2b) If a match is found then the node looks to see if the state is stored in its part of the database. If not, it sends the newly produced state to the node in which the state with the matching key is kept.

3a) The second set of keys is checked, and if successful, the states are checked, link by link.

3b) If no match is found in point 2b, the state is put on the temporary list for new state candidates.

4) After all states from the given states have been generated, nodes exchange temporary lists and remove duplicates.

5) New states are attached to the lists in processors modulo N.

6) A new state from the previous generation is chosen and the algorithm returns to point 1b.
We can estimate the efficiency of the algorithm in the similar way as we did in the previous section.

$$\frac{1}{\epsilon} = \frac{N(g\log_2 MQ/N + dQ/N + b\log_2 NQ/N + (b/4)\log_2 NQ/N)}{dQ + g\log_2 MQ} \tag{15}$$

The meaning of the variables d, b, Q, M, N in this equation is the same as in Eq. 10. The term $g\log_2 M$ represents the time of a binary search of the database in the case when the data was not divided into buckets. The last term in Eq. (15), $(b/4)\log_2 NQ/N$ is the time spent to exchange the temporary list between the nodes.

If we use the values of parameters from the previous section, then on a 32 node machine, the efficiency calculated from Eq. (15) is about 80%.

Using the general formulism described in Sec. 4, we rewrite Eq. (15) for a large database of size M in the form:

$$1/\epsilon = 1 + \frac{\gamma \log_2 N}{\log_2 M}, \tag{16}$$

$$= 1 + \frac{\gamma \log_2 N}{(\log_2 n + \log_2 N)}. \tag{16a}$$

This is in fact the same type of dependence of ϵ on N and M as seen for the Fast Fourier Transform. The logarithmic dependence on n in Eq. (16a) corresponds to an infinite system dimension d_S [Fox 85a].

We find it intriguing that in a gravitational problem, the direct N^2 formula has $d_S=1$ and the Fast Fourier Transform with similar improved $N\log N$ algorithm has $d_S = \infty$. In the database case, the simplest algorithm has $d_S=1$ again and the improved sequential algorithm corresponds to infinite system dimension.

7. Conclusions

In the future, we will be applying the algorithm to much larger databases as we study more sophisticated physics problems. Typical parameters of interest are the states that need to be generated for the 5th generation 4^3 SU(2) problem. The number of states we expect in this case is roughly 30000, and the algorithms will perform well on the hypercube.

We have shown analogies in the performance analysis of database problems to those of loosely synchronous scientific algorithms . Particularly intriguing is the analogy to N body algorithms explored in the last section. It would be interesting to explore further search algorithms to see how they fit in the performance model described in Secs. 4 and 6.

References

[Apostolakis 88] Apostolakis, J., Baillie, C. F., Ding, H. Q., Flower, J. "Lattice Gauge Theory on the Hypercube." To appear in proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, edited by G. C. Fox, published by ACM, New York, N.Y., Caltech report C^3P-605

[Flower 87] Flower, J. W. 1987 "Lattice Gauge Theory on a Parallel Computer," Caltech Ph.D. Thesis, Caltech report C^3P-411

[Fox 85a] Fox, G. 1985 "The Performance of the Caltech Hypercube in Scientific Calculations: A Preliminary Analysis." Invited Talk at Symposium on "Algorithms, Architectures and the Future of Scientific Computation," Austin, Texas, March 18-20, 1985. Published in "Supercomputers-Algorithms, Architectures and Scientific Computation," edited by F. A. Matsen and T. Tajima, University of Texas Press, Austin, 1985, Caltech report C^3P-161.

[Fox 86a] Fox, G. C., Otto, S. W. 1986 "Concurrent Computation and the Theory of Complex Systems," published in *Hypercube Multiprocessors*, 1986, edited by M. T. Heath, SIAM p. 244, Caltech report C^3P-255.

[Fox 88a] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, S. K., and Walker, D. 1988 "Solving Problems on Concurrent Processors," published by Prentice Hall 1988. The Software Supplement to the book is edited by I. Angus, G. Fox, J. Kim, and D. Walker.

[Furmanski 86] Furmanski, W and Kolawa, A. 1986 "Yang-Mills Vacuum: An Attempt at Lattice Loop Calculus," Published in *Nuclear Physics*, B291 (1987) 594-628, Caltech report C^3P-335

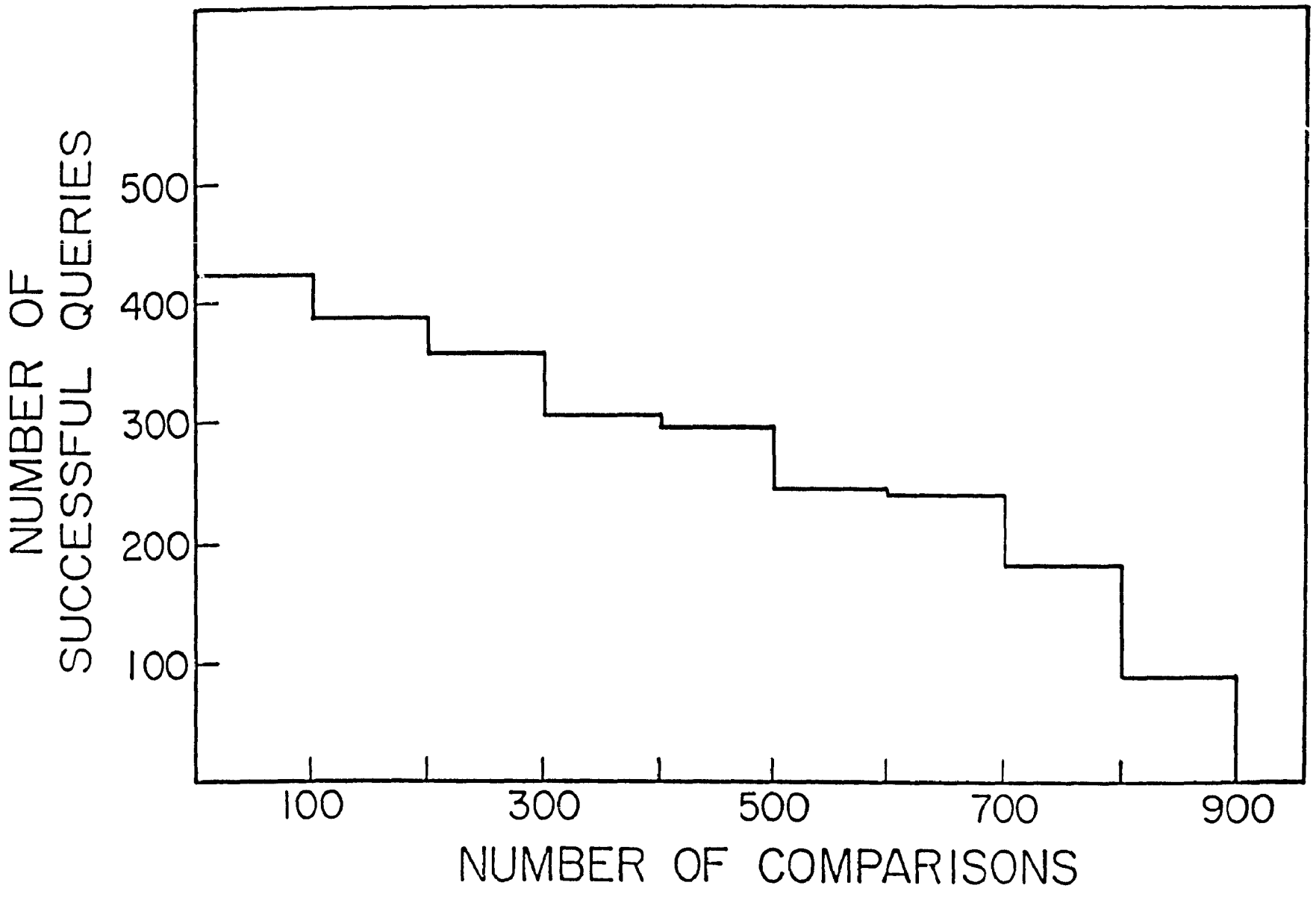
[Johnson 85] Johnson, M. 1985 "An Interrupt Driven Communication System," Caltech report C^3P-137

[Kolawa 86] Kolawa, A. and Otto, S. 1986 "Performance of the Mark II and INTEL Hypercubes." Published in *Hypercube Multiprocessors, 1986* edited by M. T. Heath, SIAM p. 272, Caltech report C^3P-254

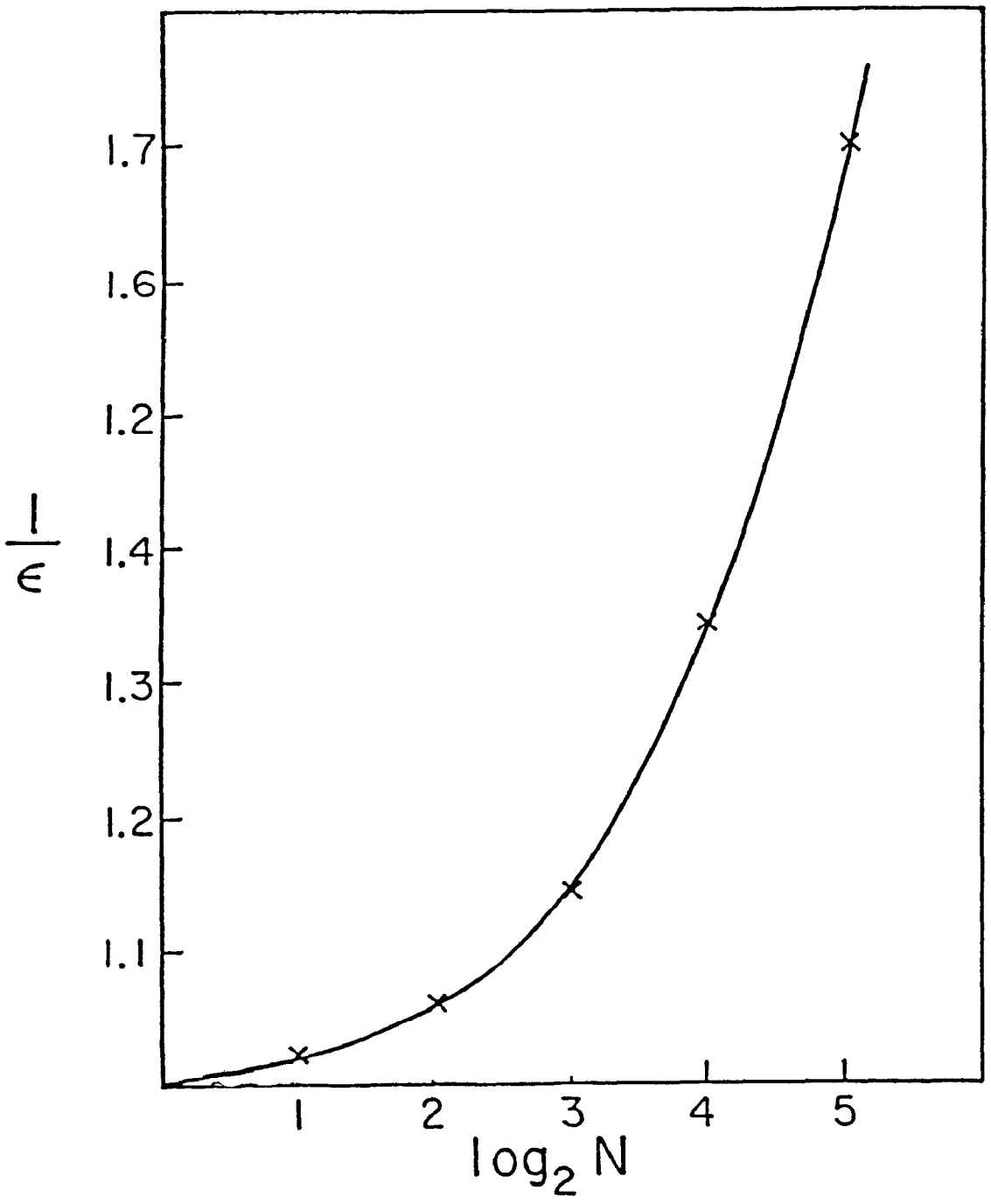
[Kolawa 86a] Kolawa, A. 1986 "Semianalytical Calculation of the O^{++} Glueball Mass in SU(2) Gauge Theory", Caltech PhD Thesis, Caltech report C^3P-267.

[Otto 84] Otto, S. and Stack, J. 1984 "The SU(3) Heavy Quark Potential with High Statistics," CALT-68-1113. *Phys. Rev. Letters* 52, 2320, Caltech report C^3P-67.

[Tuazon 85] Tuazon, J., Peterson, J., Pniel, M., Lieberman, D. 1985 "Caltech/JPL Hypercube Concurrent Processor." Published in proceedings of IEEE 1985 International Conference on Parallel Processing in St. Charles, III., (August 20-23, 1985), Caltech report C^3P-160.



1. Histogram of the number of successful queries vs. number of comparisons for sequential implementation.



2. Plot of inverse of efficiency vs. $\log_2$ of the number of nodes. The solid line is the plot of formula (10) and crosses are the measured values.