International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 2 Issue 1, January- 2013

Load Balancing using N-Queens Problem

Punita Panwar [1],Varun P. Saxena[2],Aditi Sharma [3],Vijay Kumar Sharma[4]
[1]M.Tech (p), Rajasthan Institute of Engineering & Technology Jaipur.
[2] Assistant Professor, Govt. Women Engineering College Ajmer.
[3]R.N. Modi Engineering College Kota.
[4]Rajasthan Institute of Engineering & Technology Jaipur.

***Abstract*-- A distributed system can be viewed as a collection of computing and communication resources shared by active users. These resources are distributed and possibly owned by different agents or organization. When the demand for computing power increases the load balancing problem becomes important. The purpose of load balancing is to improve the performance of a distributed system through an appropriate distribution of the application load. Load balancing is a way to keep processor utilization as even as possible. A general formulation of this problem is as follows: Given a large number of jobs, find the allocation of jobs to computers optimizing a given objective function.**

In distributed system load balancing is applied to the *N*-queen problem as the data domain is composed of *N* units, and we want to have it solved on a network of *P* processors. Our main task is applying load balancing using N-queen problem. The parallel program is set up to search for a solution containing *N* queens on an *N* by *N* chess-board by positioning a queen on successive rows, starting with the top row of the board, and going down one row at a time.

Keywords: Distributed, load balancing, performance, parallel program

I. INTRODUCTION

A. Why N- Queens Problem for Load Balancing?

THE load balancing is the process by which elements of the data domain are assigned to processors, with the same two goals of maximizing the processors' utilization, and minimizing the total execution time. . Load-balancing refers most often, to the dynamic distribution of data among the processors.

We have an application for which the data domain is composed of *N* units, and we want to have it solved on a network of *P* processors. The data domain offers great flexibility in the way it can be decomposed. Assuming that *P* divides *N* evenly, each processor can start with an equal number of data units. Let's assume, furthermore, that the processors are arranged in a chain, with *P*1 serving as a host interface. Assuming that each processor is sending intermediary results to the host as soon as it obtains them, we have a situation where *P*1 must spend a large amount of time shuffling data from the other processors in the chain to the host. This involvement in data communication affects all the processors, and lessens as we move closer to *PN*. Hence, *P*1 will spend less time processing its own data than *P*2 does, which it turns, will spend less time than *P*3, and so on, until we reach *PN* which can devote all of its time computing and processing its own data. As a result *PN* will probably finish first and run out of data before the others. We can then expect *PN*-1 to be the next to finish, and so on.

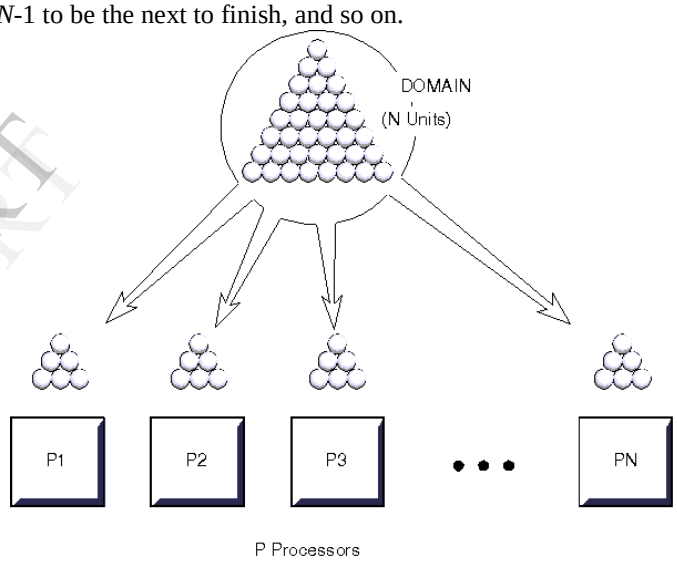

Fig: I

For this problem of load balancing here we implement a solution by *N*-queen problem.

B. What is N- Queens Problem?

The *N* queens problem is to place *N* queens on an *N* by *N* chessboard, so that no queens can take each other. Because queens can move horizontally, vertically, and diagonally, this means that there can be only one queen per row and one per column, and that no two queens can find themselves on the same diagonal. Finding a solution for a dimension *N* requires creating a search tree where each node represents a valid position of a queen on the chess board. Nodes at the first level correspond to one queen on the *N by N* board. Nodes at the second level represent boards containing two queens in valid locations, and so on. When a tree of depth *N* is found then we have solution for positioning *N* queens on the board.

Partitioning in this case can be done by assigning sub trees to the individual processors, allowing the search to be done in parallel on different sub trees. The program stops when a processor reaches a terminal leaf (success), or when all the sub trees have been visited without ever reaching a terminal leaf (no solutions). For problems that exhibit such irregular domains that grow and shrink during the computation, partitioning must often be carried out dynamically by techniques that ensure that each processor has some work to do, and that progress is made towards a solution.

The pseudo-code of the sequential solution is as follows:
beginArray () //diagonals and columns marking them empty
call to addQueen proceeding
***addQueen*()** //place a queen on the following row
row++
for each column *do*(*i*:1..*N*)
test if a queen can be placed on column *i*.
If true *then*
mark the column and diagonals as filled.
If is the last row *then*
New solution found
If not
Call *addQueen* proceeding
The solution presented above shows a non-linear growth in the complexity as the size of the board increases.

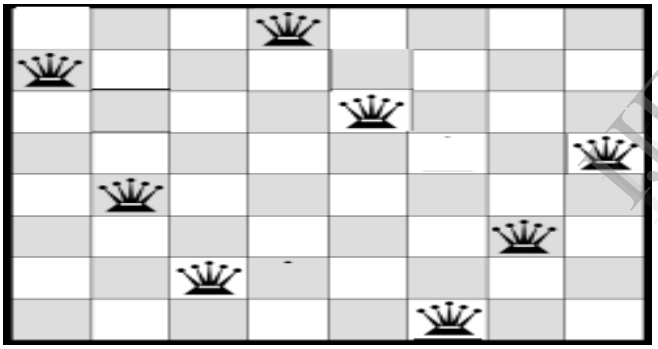

Fig: II

II. N-QUEEN PROBLEM IMPLEMENTATION FOR LOAD BALANCING

The distributed system consists of independent workstations connected usually by a local area network. Users of the system submit jobs to their computers at random times. In such a system some computers are heavily loaded while others have available processing capacity. The goal of the load distributing schema is to transfer the load at heavily loaded machines to idle computers, hence balance the load at the computers and increase the overall system performance
The parallel program is set up to search for a solution containing *N* queens on an *N* by *N* chess-board by positioning a queen on successive rows, starting with the top row of the board, and going down one row at a time. At startup, a processor is given one of the many possible starting positions of the first queen on the first row (*N* total positions exist). Each processor contains an array representing the chess-board, and from the knowledge of that first position it deduces all the allowed positions of the second queen on Row 2. Each one of these positions represents the leaf of a unary tree of height 2, and the processor records all these trees in a heap, keeping the last tree found as the current tree to which it will try to add a new level.
As the processor progresses it may find that the next row of the chess-board is completely covered by the queens already on the board, and the current tree cannot be further extended. The current tree my therefore been thrown out and a new tree must be obtained from the heap. Because the program maintains the heap as a stack, and because the program always increases the height of its current tree, the tree on top of the stack is always the taller one (other trees in the stack may have the same height). This way, when a processor has reached an impasse and gets a new tree from the heap, it gets one that has the fewest leaves to add, hence a partial solution with the highest number of queens already in place. The load in each transputer is balanced by a manager-workers scheme, where the root transputer hosts the manager. The manager accesses its workers via a virtual star network (network of virtual channels) that can be mapped over any physical network.
The Manager-Workers paradigm is easy to implement. Virtual channels allow the creation of a star-shaped network on any physical network with very little effort. The Manager sits at the center of the star and enjoys direct access to the workers. For larger networks, however, this simplicity may not be acceptable due to the delays that may dramatically reduce the processors' utilization. In such cases, distributed methods for load-balancing may be more attractive, and because they require a synchronization that involves only near neighbors, lower performance penalty can be expected in general. In addition, the processor originally implementing the Master can now be given a bigger share of the computation.

Algorithm Load-Unbalance N-queens (Metrics C_1, Metrices C_2, queen I_i)

1. Maximum work load L_{max} and minimum work load L_{min} of the processors.
   If L_{max}=1000 and L_{min}=500

2. Unbalance the load in between L_{max} and L_{min}, Metrics is defined as-

   $$C_1 = L_{max} / L_{min}$$

   $$C_1 = 1000/500$$
   ⇨ $C_1 = 2$

3. Average work done is-

   $$W = \sum_{Ti=1}^{N} |Ti - Tprom|/N$$

4. Unbalance related to the average work done, Metrics C_2 into account the deviation percentage of the work done by the processors in relation to the average of the work done-

$$C_2 = (W * 100) / T_{prom}$$

If C_2= 0 then the obtained balance is the optimal.
If C2 = 50 it means that each processor deviates a 50 % of the work that it should carry out if it had an optimal balance.
The study of the load unbalance has been initiated for a type of parallel systems, focusing on the adjustment of the algorithm to the supporting architecture for load balancing.

III. Parallel Algorithm for load balancing N-Queen Problem

The parallel algorithm for the same is intuitive. Assign first configuration of the queens to each node and run the sequential program on the same. If $p < n/2$ then each node will attain a maximum of n/2p initial configurations.
Each node with an initial configuration will run a Depth First Search. Any state cannot be further expanded if there are no non-conflicting positions for the queen in the next column and the program would need to back-track.
Manager-worker paradigm has been invoked for this parallelization. Node with rank 0 is assumed to be the manager while the rest are workers. The workflow is as follows:

1. Manager waits for request for work. A worker sends request for work if it's idle.
2. Manager assigns tasks to the requesting worker.
3. Manager waits for another request until no more work.
4. On assignment of a task, the worker will process it and send back all the results.

IV. Results

For our purpose, tuning can be applied to finding the best interval of time between load-balancing periods. Hence the performance of the load-balanced application was dependent on the heuristic used to balance the load, and on the update interval. Because the Robin-Hood heuristic we choose here always brings back the two nodes that support the extreme loads, it must be run often to make sure the gap between these nodes (possibly different every update) does not increase. Tuning can take the form of a simple experiment where the parameter of interest is defined at *run time* of the *N*-queen problem for *N*=27 and various interval lengths.

For board size = 12 using 7 processors: Got 7 slots.
Node = 1: Results computed = 2139
Node = 5: Results computed = 2330
Node = 3: Results computed = 2139
Node = 6: Results computed = 2718
Node = 2: Results computed = 2437
Node = 4: Results computed = 2437
Total number of Results = 14200

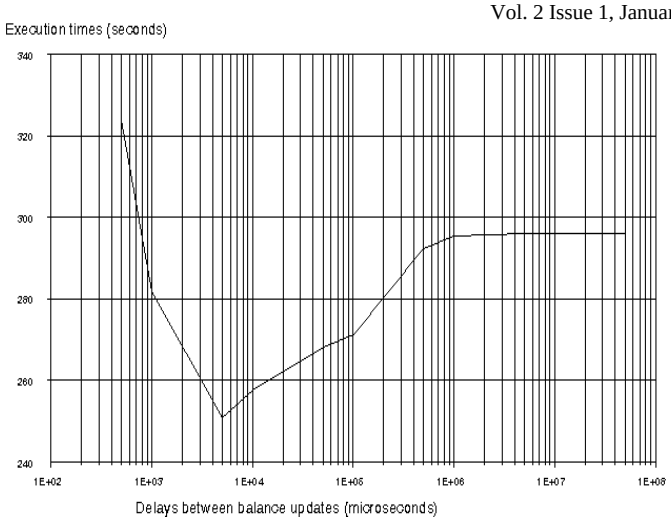

Fig: III

Tuning the N*-queen problem for* N*=27, and intervals ranging from 500 sec. to 50 sec. Each execution time is the average of two runs.*

Similarly for board size = 15 using 8 processors:
Node = 2: Results computed = 357770
Node = 7: Results computed = 304450
Node = 6: Results computed = 323927
Node = 4: Results computed = 323927
Node = 5: Results computed = 332330
Node = 3: Results computed = 304450
Node = 1: Results computed = 332330
Total number of Results = 2279184

V. Conclusion

In this paper a fast and practical approach of load balancing for the N-queens problem is used. By using n processors, a5 ms interval length provides the best execution time, 250.94 seconds, compared to 295.89 seconds for the asymptotic bounds for longer delays.

VI. References

[1] **INMO88b** Inmost, *IMST800 Transputer*, Document No. 42 1082 00, March 1988.
[2] **BRAW89** Brawer S. *Introduction to Parallel Programming.* San Diego, CA: Academic Press, Harcourt Brace Jovanich, Publishers, 1989.
[3] Parallel Programming in C for the Transputer © *D. Thiébaut, 1995*
[4] **AMDA67** Amdahl, G. M. "Validity of the single-processor approach to achieving large scale computing capabilities," *AFIPS Conf. Proc. 30*, AFIPS Press: 483-485, 1967.
[5] **DALL92** Dally W. J. "Virtual-channel flow control," *IEEE Trans. on Parallel and Dist. Syst.*, 3(2):194-205, March 1992.
[6] **EAGE89** Eager D. L., J. Zahorjan, and E. D. Lazowska "Speedups versus efficiency in parallel systems," *IEEE Trans. Computers* 38:406-423, Mar 1989.
[7] **ELM86** Elmagarnid A. K. "A survey of distributed deadlock detection algorithms," *SIGMOD Records*, 15(3):37-45, Sept. 1986.
[8] **FEIT91** Feitelson D. "Deadlock detection without wait-for graphs," *Parallel Computing*, 17:1377-1383, 1991.
[9] **FLYN72** Flynn, M., "Some computer organizations and their effectiveness," *IEEE Trans. Comp.* 21(9):948-960, 1972.

[10] **GOSH91** Goshal D, G. Serazzi, and S. Triphath "The processor working set and its use in scheduling multiprocessor systems," *IEEE Trans. Soft. Eng.*, 17(5):443-453, May 1991.
[11] **GRØN91** Grønning P., T. Qvist Nielson, and H. H. Løvengreen "Stepwise development of a distributed load balancing algorithm," *Lecture Notes in Comp. Science*, (486):151-168, 1991.
[12] **GUNT81** Gunther K. D. "Prevention of deadlocks in packet-switched data transport systems," *IEEE Trans. Commun.*, COM-29:512-524, Apr. 1981.
[13] **INMO88b** Inmos, *IMST800 Transputer*, Document No. 42 1082 00, March 1988.
[14] **INT86** Intel, *Introduction to the 80386*. 231746-001, April 1986.
[15] **KNAP88** Knapp E. "Deadlock detection in distributed databases," *Technical Report*, Depat. Computer Science, University of Texas, Jan. 1988.
[16] **KROG91** Kröger B., R. Lüling, B. Monien, and O. Vorngerger. "An improved algorithm to detect communication deadlocks in distributed systems," in *Lecture Notes in Computer Science*, 486:90-101.
[17] **KUNG88** Kung H. T. *VLSI Array Processors.* Englewood Cliffs, NJ: Prentice Hall, 1988.
[18] **LEST93** Lester B. P. *The art of Parallel Programming*, Englewood Cliffs, NJ: Prentice Hall, 1993.
[19] **THIE92** Thiébaut D. J. Wolf, and H. Stone, "Improving disk-cache performance with partitioning" *IEEE Trans. Computers*, 41(6), June 1992.
[20] **VONN45** Von Neumann, J. "First draft of a report on the EDVAC" Moore School, University of Pennsylvania, 1945.
[21] **WILS**, Wilson P. "Highly concurrent systems using the transputer," *Tech. Report*, Inmos Corporation.
[22] Adaptive Load Balancing for MPI Programs by Milind Bhandarker, L.V. Kale, Eric De Sturler, Jay Hoeinger
[23] Dynamic Load Balancing of Unbalanced Computations Using Message Passing by James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng
[24] Common Search Strategies and Heuristics With Respect to the n-queens
[25] Problem by Sheldon Dealy