VFAST Transactions on Mathematics
http://vfast.org/journals/index.php/VTM@ 2020, ISSN(e): 2309-0022, ISSN(p): 2411-6343
Volume 8, Number 1, January-December, 2020 *pp: 45-53*

IMPLEMENTATION OF DISCRETE SEARCH OPTIMIZATION ALGORITHMS IN PARALLEL AND THEIR PERFORMANCE ANALYSIS

Mohammad Zulqurnain[*] AnilaUsman[*]
Muhammad Hanif Durad[*] and Idrees Ahmad[$]
[*]Department of Computer and Information Sciences,
[$]Department of Nuclear Engineering
Pakistan Institute of Engineering and Applied Sciences (PIEAS), Islamabad, Pakistan
{prodigiousguy, anila, hanif and idrees } @hotmail.com

Abstract: The present paper discusses the implementation of the discrete search optimization techniques on parallel platform (SGI-Altix 450 shared memory 64 processors system). We show that the combination of Asynchronous Parallel Iterative Deepening and Parallel Window Search technique tends to give more challenging speedups, memory consumption and efficiency with less resources consumption as compared to the rest of the techniques. In general 5 techniques are compared with Parallel Asynchronous Window Search technique among which includes Depth First Search, Parallel Iterative Deepening, Parallel A*, Parallel Window Search and Parallel Asynchronous Iterative Deepening A*.
Keywords: Discrete combinatorial problems, MPI, C++, Linux, MPE performance profiling, combinatorial discrete-optimization

1. **Introduction.** A discrete optimization problem (DOP) can be expressed as a tuple (S, f). The set S is a finite or countably infinite set of all solutions that satisfy specified constraints. This set is called the set of feasible solutions. The function f is the cost function that maps each element in set S onto the set of real numbers R. f: S R. The objective of a DOP is to find a feasible solution xopt, such that f (xopt) f (x) for all x €S. Problems from various domains can be formulated as DOPs. Some examples are planning and scheduling, the optimal layout of VLSI chips, robot motion planning, test-pattern generation for digital circuits, and logistics and control.[12] Heuristic search provides the driving force for many applications of artificial intelligence (AI) including problem solving, robot motion planning, concept learning, theorem proving and natural language understanding. However computational complexity is major limitation of search, thus research community is continually trying to develop more efficient search algorithms. Parallel search algorithms significantly increase the size of the search space that can be traversed in a given amount of time. Parallel search techniques have been implemented on MIMD architectures. Due to the overwhelming size of real-world AI applications, and because of the increasing power and accessibility of parallel computers, there exists a constant need for improvement of parallel search algorithms. Heuristic search focuses on using information about the problem to guide the search. In the fifteen puzzle, for example, well known heuristic can be used to estimate the maximum number of moves needed to get to a known goal from any board. Using this estimator, search techniques can be designed that will produce an optimal solution, while searching only a small portion of the search space. We have compared five other parallel search techniques with our search technique in this paper in solving 15-puzzle which is an NP-hard discrete optimization problem. The paper is divided into five sections: section II gives an overview of search techniques, section III describes the new technique, Parallel Asynchronous Window Search which is basically the hybrid of the two techniques. Results are shown in section IV and conclusion and future work are given in sections V and VI.

2. **Overview of Search techniques.** IDA* is a mechanism which involves the concept of iterative heuristic search in which heuristic bound is eventually iterated and cut-off bound is saved each time for the next iteration. Following is the algorithm for IDA*:

Procedure IDA(n)*
Bound:=h(n);
While not solved do
Bound:=DFS(n,bound);
Function DFS(n,bound);
If f(n)> bound
    then return f(n)
If h(n) =0
    then return solved;
return lowest value of DFS(n,bound)
    for all successors ni of n

Where i denotes the heuristics of the children of the minimum heuristic node [4][10]. This algorithm is used in the following techniques

A. Parallel formulation of IDA (PIDA*):*

There are 3 variations to IDA* parallel implementation, first one is SIDA* which was implemented by Korf et al. The outline for this work is divided into 3 steps, the initial node distribution by the master processor to the rest of the processors and then each processor carries out IDA individually on its node (i.e. subtree), in case there is a load imbalance, dynamic load balancing takes place in which stack of the respective processors is split and the work is shared amongst the neighboring processors. The biggest disadvantage of this technique was that its stack load balancing scheme took too much time and despite of that was still inefficient since the subtrees possessed by subsequent processors after the dynamic load balancing step were still uneven due to which the scalability of this technique is limited[1][2][3].

Time complexity : $t_{stack\ split\ over}(P) = O(logP*P^2)$

B. Asynchronous iterative depth first A(AIDA*)*

Asynchronous iterative deepening A* search algorithm was devised by Reinefeld et al [7][8][11].

Basically this algorithm was divided into 3 phases:-

1. A short initial data partitioning phase, where all processors redundantly expand the first few tree levels in an iterative-deepening manner until a sufficient amount of nodes is generated to keep each processor busy in the next phase,
2. An additional distributed node expansion phase, where each processor expands its ‘own’ nodes of the first phase to generate a larger set of, say, some thousand fine grained work packets for the subsequent asynchronous search phase,
3. An asynchronous search phase, where the processors generate and explore different subtrees in an iterative-deepening manner until one or all solutions are found.

Time complexity :

$t_{fixed\text{-}pack\ over}(P)= O(P)+P*t$

$t_{last\ phase}(P)=O(P^2)$

Window iterative deepening A*(WIDA*)[4]

It is composed of two steps:

- Initial node partitioning:. The idea ordering scheme is to fully exploit the information available from the most recently completed frontier. This can be done by saving the entire set of leaf nodes from the most recent iteration, and then ordering them for expansion by h minimum h [5].

- Window search. When the search space's entire frontier-set has been searched to its threshold, the process broadcasts its ordering information to all the other processes. The message consists of (1) minimum h value for each of the frontier-set nodes, and the associated path, and (2) the value of the threshold on which the ordering information is based. On receiving message each processor aborts its current search and restarts it with the updated heuristic. On competing messages only the message with the most deepest depth is taken into account [5].

Time complexity : $O(B^{i} + B^{I-i})$ \

Where B denotes the search Bound i.e. local heuristic-depth of the node, I denotes maximum heuristic which is 64 in the case of 15-puzzle and i denotes the total number of processors.

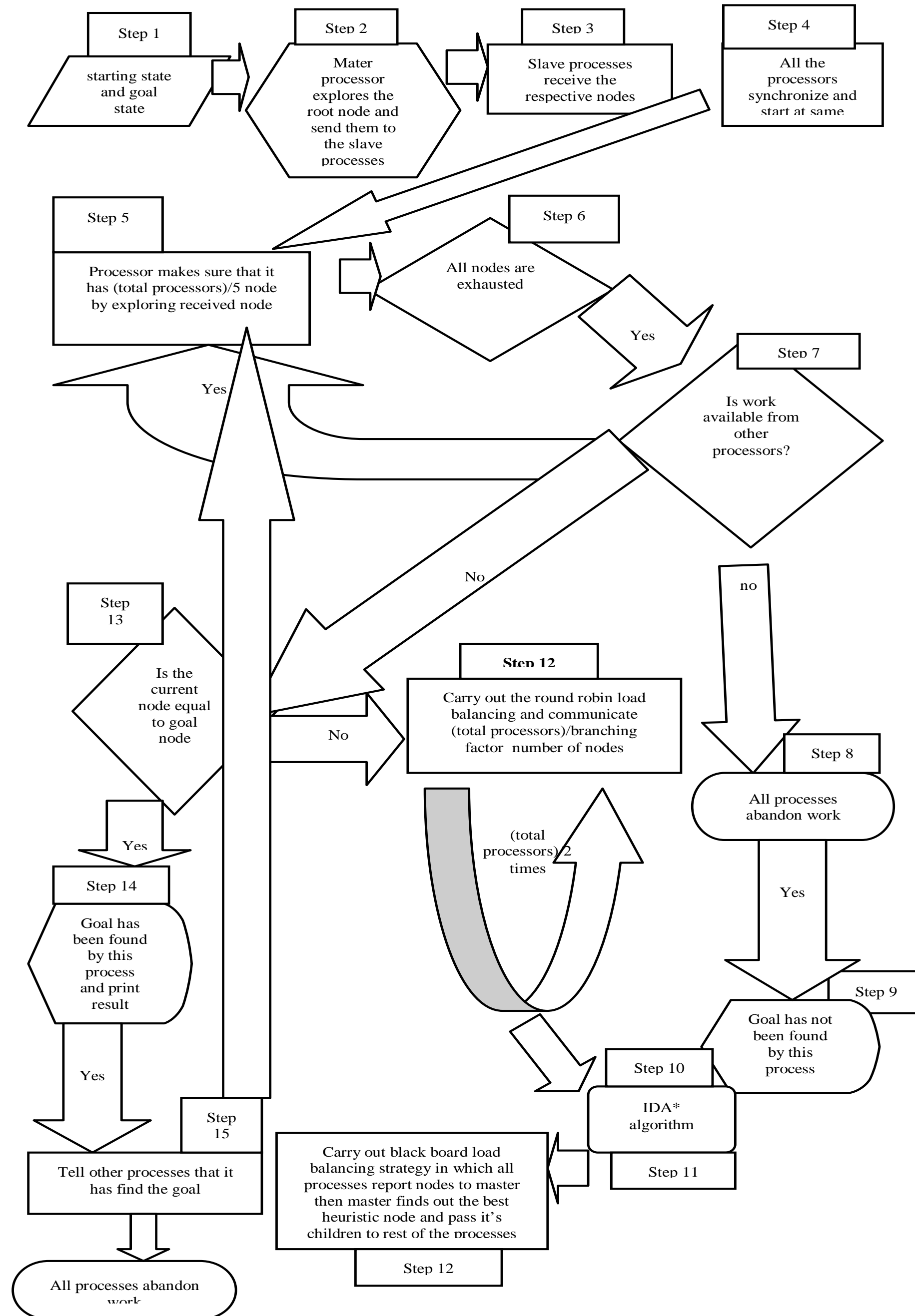

Figure 1: PAWS architecture

3. Parallel Asynchronous window search. In our technique we have utilized the space optimality from IDA* algorithm. We have taken optimal load balancing from the fixed packet load balancing attribute of AIDA* and Optimal node ordering from WIDA*.

A. Architecture

Following is the architecture for our PAWS system which gives the overview of the communication and computation carried out by the processors.

B. Overall computational complexity

In time complexity the first expression adheres to starting time which is step 2, the second expression adheres to initial node generation which is step 5, the third pertains to load-balancing scheme i.e. round robin which is step 12, the fourth to computation with respect to the window heuristic bound partitioning concept which is step 11 and the fifth to the load balancing step in which the idle processors gets work from the owner of the best heuristic bound node which is step 7. In the long run the third expression dominates i.e. in the case of fifteen puzzle the worst heuristic is M_h =15*15=225 and if we increase number of processors above this threshold then time complexity will be proportional to P^2 *logP.

b denotes branching factor, i denotes maximum manhatten heuristic possible, d denotes the depth of the node, P denotes number of processors, W denote problem size and M_h (in 15-puzzle case it is 16) denotes the maximum disposition of a tile.

Within the time complexity term the first term represents starting time, the second term denotes initial data partitioning, the third term is for round robin load balancing, the fourth term is for window search heuristic window partitioning step and the last term is for djikstra's termination detection algorithm.

Time complexity:

$$O(logP)+O(P/20)+O((P/4)^2logP+ O(M_h-P)+O(P)$$

Speedup:

$$P*(bd)/[(P*logP)+(P^2/20)+((P)^3/4logP)+O[P(M_h -P)] +P^2]$$

Efficiency:

$$(bd)/[(P*logP)+(P^2/20)+((P)^3/4logP) +O[P(M_h -P)] +P^2]$$

Space complexity:

$$O(P*b*d)$$

Isoefficiency:

$$W=O(P^2logW)$$

$$W=O(P^2log(P^2logW)+P^2loglogW)$$

The double log is asymptotically smaller than the first term, provided P grows slowly no lower than logW, and can be ignored. The isoefficiency function for this scheme is therefore given by $O(P^2logP)$.

4. Results. Taking Korf's[1] 100 random 15-puzzle problem instances and breaking them in to four classes C1,C2,C3 and C4 we have implemented the 50 hardest problems of the classes C3 and C4 on our SGI-Altix 450 super computer with 64 processors.

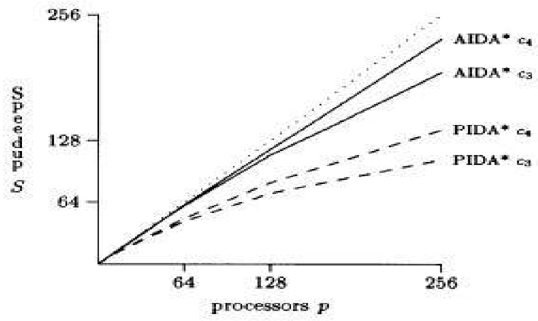

Figure 2: speedup VS number of processors obtained by Reinefeld and V.Schnecke[11]

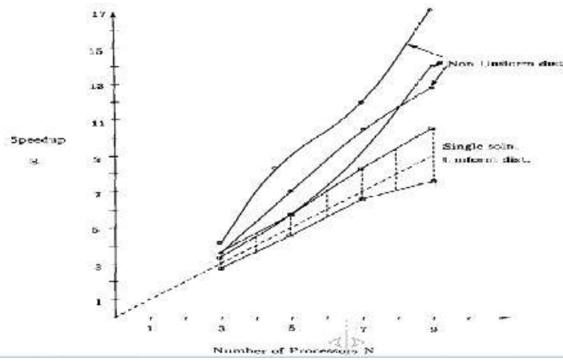

Figure 3: speedup vs number of processors attained[2] by Rao et al

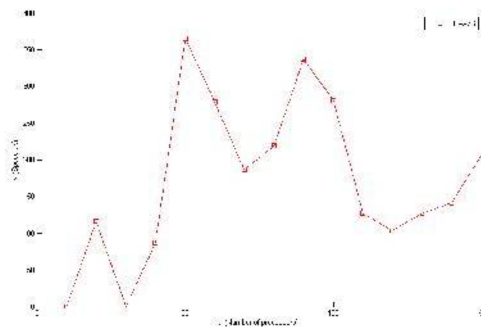

Figure 4: Speedup vs number of processors (PAWS)

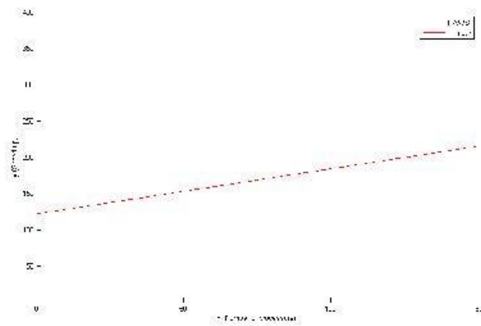

Figure 5: speedup vs number of processors regression line

A. *Performance profiling*

In the Time line profile diagram of PAWS the red color represents the messages being exchanged by the processors, Mustard shows the initial load distribution by the master processor and purple represents the receive by the processes, Yellow represent dark Pink, black color represent the window search and black board balancing load balancing scheme, light pink represents the phase in which each processor is carrying out initial node partitioning to generate size/5 nodes each, aqua blue color shows setting of receiver of the goal from the processor who has found it, golden shows the sending of goal node from the successful processor to other processors, and dark green color shows the Djikstra's termination detection carried out by the processors.

Table 1: Speed-up table for PAWS

Number of processors	Speed-up for Parallel Asynchronous Iterative Window search	Efficiency for Parallel Asynchronous Iterative Window search
10	0.0219	0.0021
20	116.2	5.817
30	0.3987	0.0133
40	87.25	2.1812
50	365.4958	7.3099
60	278.4730	4.6412
70	186.6362	2.6662
80	219.2975	2.7412
90	337.3808	3.7487
100	280.7008	2.8070
110	127.5913	1.1599
120	103.1988	0.8599
130	127.5913	0.9815
140	141.4823	1.0106
150	206.3977	1.38

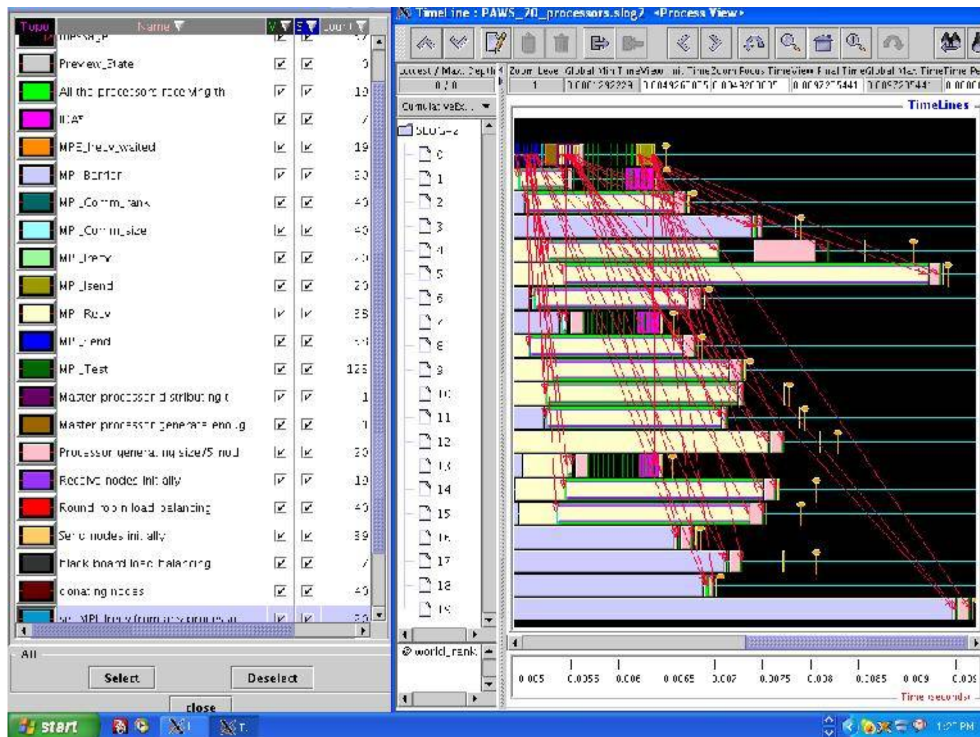

Figure 6: Timeline profile of PAWS for 20 processors

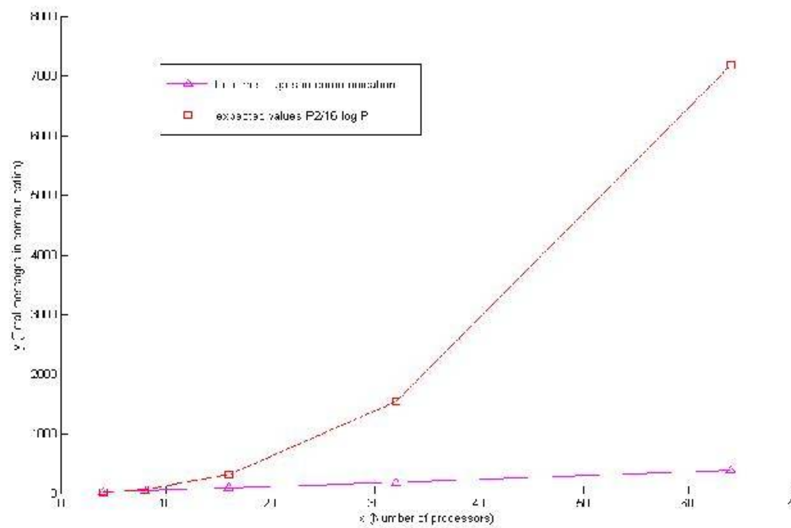

Figure 7: messages per processor generated in PAWS

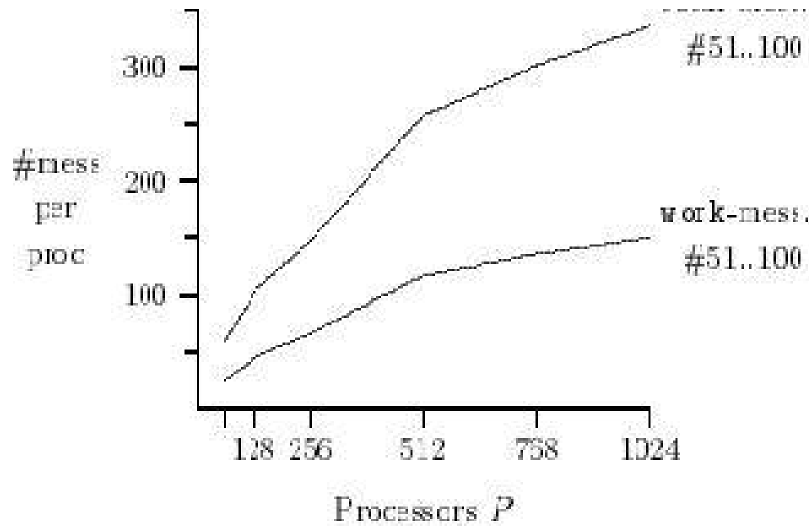

Figure 8: Messages per processor [7]

5. Conclusion. Our technique has performed better than other parallel versions of IDA* algorithms in terms of speed-ups. This is mainly due to the pure window search node ordering concept that we have used in this algorithm which harness greater parallelism in which different processors work at different heuristic bounds of the search space and finds goal node earlier than the conventional IDA* algorithm, secondly this algorithm is asynchronous in nature that is every processors pursues IDA* on its best node after extracting certain number of nodes i.e. communication overhead is minimum in this phase and thirdly in the load balancing phase in which fixed sized chunks are traded by each processor to its neighbor in a ring topology i.e. it is made sure that each processor posses fairly equal sized sub-tree. Memory consumption of our technique is also better than the previous versions of parallel IDA*. Our algorithm is more scalable as compared to the other counter parts of IDA* due to its loose synchronous nature and parallel window node ordering aspects. Furthermore our communication analysis tells us that PAWS is more asynchronous as compared to AIDA*[7].

6. Future work. Our future work will comprise of application of our algorithm to other discrete optimization problems besides 15-puzzle problem which includes VLSI floor planning problem, hackers problem, travelling salesman problem etc.

REFERENCES

[1] Cook, D. J., & Lyons, G. (1993). Massively parallel IDA* search. *International Journal on Artificial Intelligence Tools*, *2*(02), 163-180.

[2] Rao, V. N., Kumar, V., & Ramesh, K. (1987). *A parallel implementation of iterative-deepening-A* (pp. 178-182). Artificial Intelligence Laboratory, University of Texas at Austin.

[3] Evett, M., Hendler, J., Mahanti, A., & Nau, D. (1995). PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, *25*(2), 133-143.

[4] Mahanti, A., & Daniels, C. J. (1991). Simd parallel heuristic search.

[5] Nerur, S., & Cook, D. J. (1994, May). A hybrid parallel-window/distributed tree algorithm for improving the performance of search-related tasks. In *Proceedings of the 7th international conference on Industrial and engineering applications of artificial intelligence and expert systems* (pp. 629-637).

[6] Fenton, W., Rankumar, B., Salctore, V. A., Sinha, A. B., & Kale, L. V. (1991, August). Supporting machine independent parallel programming on diverse architectures. In *proceedings of 1991 International Conference on Parallel Processing, pp. II-193-201, Boca Raton, Florida*.

[7] Reinefeld, A., & Schnecke, V. (1994, May). AIDA^*-Asynchronous Parallel IDA^. In *Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence* (pp. 295-302). Canadian Information Processing Society.

[8] Reinefeld, A., & Marsland, T. A. (1994). Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *16*(7), 701-710.

[9] Ibrahim, M. A., Xinda, L., & Rwakarambi, J. M. (2001, January). Parallel execution of an irregular algorithm depth first search (DFS) on heterogeneous clusters of workstation. In *2001 International Conferences on Info-Tech and Info-Net. Proceedings (Cat. No. 01EX479)* (Vol. 3, pp. 328-332). IEEE.

[10] Powley, C., & Korf, R. E. (1991). Single-agent parallel window search. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (5), 466-477.

[11] Reinefeld, A., & Schnecke, V. (1994, May). Work-load balancing in highly parallel depth-first search. In *Proceedings of IEEE Scalable High Performance Computing Conference* (pp. 773-780). IEEE.

[12] Grama, A., Kumar, V., Gupta, A., & Karypis, G. (2003). *Introduction to parallel computing*. Pearson Education.