Parallel Iterative .A* Search : An Admissible Distributed Heuristic Search Algorithm

Shie-rei Huang
Artificial Intelligence Department
FMC Corporate Technology Center
1205 Coleman Ave., Box 580
Santa Clara, CA 95052

Larry S. Davis
Computer Vision Laboratory
Center for Automation Research
University of Maryland, College Park
College Park, MD 20742

Abstract

In this paper, a distributed heuristic search algorithm is presented. We show that the algorithm is admissible and give an informal analysis of its load balancing, scalability, and speedup. A flow-shop scheduling problem has been implemented on a BBN Butterfly Multicomputer using up to 80 processors to empirically test this algorithm. From our experiments, this algorithm is capable of achieving almost linear speedup on a large number of processors with a relatively small problem size.

1 Introduction

Best-first heuristic search algorithms, such as the .A* algorithm, are one of the most important techniques used to solve many problems in artificial intelligence and opera! ions research. A common feature of heuristic search is its high computational complexity, which has significantly limited its application in practical domains such as flexible manufacturing, strategic planning, and management.

In the past decade, many parallel architectures have been proposed and some of them are now commercially available. Advances in parallel computer technology have offered the potential to greatly speedup the computations in general. Due to the combinatorial aspects of heuristic search, a very large scale of parallelism can be potentially explored. However, the efficiency of heuristic search algorithms mainly comes from the intelligent guidance of heuristics. When implementing parallel heuristic search algorithms on commercial multicomputers, researchers often face a tradeoff between the faithfulness to global heuristics and the high communication cost which serializes and slows down the computation. Until recently, there was no easy solution to this dilemma.

In this paper, we present a distributed best-first heuristic search algorithm, *Parallel Iterative A* (PIA*)*. We show that the algorithm is admissible, and we give an informal analysis of its load balancing, scalability and speedup. To empirically test the *PIA** algorithm, a flow-shop scheduling problem has been implemented on the BBN Butterfly Multicomputer [BBN, 1985] using up to 80 processors. From our experiments, this algorithm is capable of achieving almost linear speedup on a large number of processors with relatively small problem size.

We will assume the reader is familiar with the *A** algorithm [Hart *et al*., 19(58].

1.1 Related Work

Research in parallel heuristic search has been very active recently. *A** can be parallelized by storing the OPEN list in global storage that is accessible to all processors [Mohan, 1982]. Huang and Davis [1987] use queueing theory to show that this approach can achieve almost linear speedup to a certain number of processors. However, beyond that point, the speedup levels off suddenly, no matter how many processor are used. Rao and Kumar [1988] proposed a concurrent heap data structure for organizing the global OPEN list. The new data structure allows processors to interleave operations on OPEN and improves the speedup to some extent; however, congestion near the root of the concurrent heap is still a problem.

Kumar *et al.* [1988] propose another approach which substitutes a shared BLACKBOARD for the global OPEN list and let each processor maintain its own local OPEN list, Unfortunately, the BLACKBOARD may eventually become a bottleneck as the number of processors increases.

A distributed approach has also been tried by some researchers. Early work is represented by Wah and Eva Ma's MANIP [1981]. Anderson and Chen [1987] presented an algorithm to perform distributed best-first search on hypercube multicomputers. To balance the workload, they proposed to exchange a *summary* of the cost distribution in the local OPEN lists of neighboring processors. Quinn [1987] presented four other implementations of the best-first search on hypercubes. These four simple algorithms tried to either improve the useful computation at each processor, or to improve the communication cost, but failed to effectively improve both at the same time.

2 The *PIA** Algorithm

*PIA** proceeds by repetitive synchronized iterations. At each iteration, processors are synchronized twice to carry out two different procedures: the *node expansion procedure* and the *node transfer procedure.* Operations are largely local to the processor in the node expansion

procedure and are completely local in the node transfer procedure. Data structures in *PIA** are distributed to avoid bottlenecks. Node selection, node expansion, node ordering and successor distribution operations are fully parallelized by processors. Processors performing searches which are not following the current best heuristics are synchronized to stop as soon as possible to reduce search overhead (the increase in the number of nodes that must be expanded owing to the introduction of parallelism). During processor synchronization, speculative computations are contingently performed at each processor, trying to keep processors always productively busy, to reduce synchronization overhead. Unnecessary communications are avoided as long as processors are performing worthwhile search work to reduce communication overhead. A symmetric successor node distribution method is used to direct load balancing. Finally, the correct termination of *PIA** is facilitated by its iterative structure.

A general scalable parallel architecture model is used by us to describe *P1 A**. The architecture consists of a set of processor-memory pairs which communicate through an unspecified communication channel. The communication channel can be realized using a shared memory or by message parsing. Memory referencing through local memory is completed in constant unit time. A remote reference through the communication channel, however, requires *()(logP)* time under normal balanced traffic, where *P* is the number of processors. It is our belief that this architecture model is general enough to subsume most scalable multicomputers which are currently available commercially, such as the BBN Butterfly [BBN, 1985], the Intel Hypercube [Intel, J986] and the Connection Machine [Ilillis, 1985].

Because some forward references are required for us to describe *PIA*,* the reader may need to re-read this section to understand this algorithm.

2.1 Data Structures

In each processor j, two lists are maintained in its local memory: the work list (WL_j) and the reception list (RL_j), as shown in Figure 1. WL_j is a priority queue and RL_j is a simple list. At the beginning of each iteration, WL_j contains the sorted nodes awaiting expansion by processor j, and $RL_j = \emptyset$. During the node expansion procedure, successor nodes generated are distributed to RL_j, $j = 0 \cdots P-1$, using a successor distribution algorithm to be described in Section 2.5.

Henceforth, we will use WL_j^i to denote the set of nodes in the work list of processor j at the beginning of iteration i. If the subscript is omitted, it means the union of all P processors' work lists. That is,

$$WL^i = \bigcup_{j=0}^{P-1} WL_j{}^i.$$

If the superscript is omitted, it means for all iterations. Similar notation will be used throughout this paper.

Initially, $WL^0 = \{s\}$ and $RL^0 = \emptyset$. As PIA^* proceeds, WL^i is similar to a snapshot of the OPEN list in A^*, but distributed.

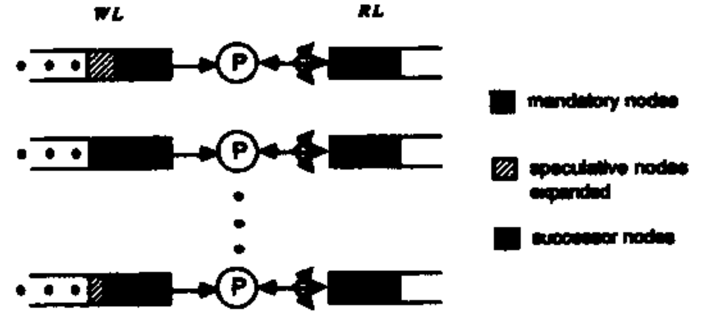

Figure 1: Data Structures and Data Flow in PIA^*

2.2 Iteration Threshold, Mandatory Nodes and Speculative Nodes

A threshold t^i is associated with each iteration i, with $t^0 = h(s)$. How t^{i+1} is actually computed will be described in the next section.

A node $n \in WL^i$ is defined to be a ***mandatory node*** if $f(n) \leq t^i$. Let M^i, which is a subset of WL^i, be the set of all mandatory nodes. As we shall see later, all mandatory nodes will eventually be selected for expansion by either A^* or PIA^* in the worst case. A node $n \in WL^i$ is defined to be a ***speculative node*** if $f(n) > t^i$. Let S^i be the set of all speculative nodes. Then, $WL^i = M^i \cup S^i$.

Initially, $M^0 = \{s\}$ and $S^0 = \emptyset$.

2.3 The Node Expansion Procedure

The node expansion procedure at each iteration *i* operates as follows. A processor *j* first expands all the nodes from M_j^i and, by an algorithm to be described in Section 2.5, puts all the successor nodes generated into the reception lists *(RL)*. Then, as long as any other processor is expanding a mandatory node, this processor continues to expand the best speculative node from *Sj'*. When all the nodes from M' have been expanded, all processors synchronize for the node transfer procedure which is described in Section 2.6.

In *PLA* successors generated by a node expansion are not considered for expansion until the next iteration; they are added to *RL* immediately after they are generated.

The node expansion procedure also computes /'+1 and broadcasts it to all processors. It is set to the maximum of (a)/7, and (b)the minimum cost of (i)all successors generated from nodes in M^i and (ii)the nodes in .S". Let *Suc(-)* be an operator which maps a set of nodes to their successor nodes. Then, f' +1 can be expressed as:

$$t^{i+1} = max(t^i, min\{f(n) \mid n \in Suc(M^i) \cup S^i\}). \quad \text{(Eq1)}$$

There is an efficient parallel method for computing /'+1. For each processor j, a local constant *Cj* which is the minimum cost of (a)all successors generated from nodes in *Mj* and (h)the best node in *Sj* can be computed during the node expansion procedure. Then, the minimum of *Cj, j* = 0 P — I, can be computed in parallel in time O(logP) [Paige and Kruscal, 1985] while processors are synchronized; /'+1 is then the maximum off and the computed minimum.

2.4 Synchronization

Let us digress here to discuss how processor synchronization in the node expansion procedure can be efficiently and correctly implemented. Essentially, what we need is a *barrier synchronization* between processors. All of the processors are required to meet at the barrier before any are allowed to proceed. The barrier in our case is a state in which each processor *j* has finished expanding *Mj.* But, instead of unproductive waiting at the barrier, a processor *j* continues to expand the best speculative node from S_j while waiting.

If a shared memory is available, the barrier synchronization can be implemented by having a global variable which counts the number of processors that are waiting at the barrier. When all of the processors have arrived at the barrier, the barrier can be removed. I his approach requires atomic operations on a global variable and may create a hot spot.

A better approach, called *the butterfly barrier* suggested by Brooks III [198(5], is to let each processor synchronize with another processor pairwise at each of log *P* stages in order to synchronize *P* processors. This approach removes the critical regions and hot spots with desired scalability, and is suitable for message-passing architectures, such as hypercubes, as well.

2.5 The Successor Distribution Algorithm

Successor nodes generated by a processor are put into RL_j, $j = 0$ $P — 1$, in a *multiplex round-robin* fashion. More precisely, suppose that the most recent successor node generated by processor *j* is added to RL_j. Then the next successor node generated by processor *j* will be added to RL_k, when* *k* — (i+ 1 *mod P).* At each iteration, processor *j* sends its first generated successor to RL_j.

The advantage of this approach is that it is simple to implement and its symmetric structure helps *PLI* attain the desired load balancing (see Section 5.1). Since the successors generated are not considered for expansion until the next iteration, some optimization can be made for message-passing architectures. Messages for successor distribution can be asynchronous so that computation and communication can be overlapped. For architectures which require large communication setup time, successor nodes generated can be distributed and cached in local memory and not sent until an efficient message size for the underlying architecture is reached.

2.6 The Node Transfer Procedure

After the node expansion procedure, each processor *j* empties the nodes from RL_j and inserts them into II L_j., to form a new priority queue for the next iteration. Note that the node transfer procedure is completely local; no communications between processors are required.

The relationship between WL^{i+1} and WL^i can be expressed as:

$$WL^{i+1} = (WL^i - M^i - \Sigma^i) \cup Suc(M^i \cup \Sigma^i). \quad \text{(Eq2)}$$

where $\Sigma^i \in S^i$ is the set of speculative nodes which were selected for expansion by processors in the node expansion procedure at iteration i to use the otherwise idle time for processor synchronization.

2.7 Termination

When a mandatory node is found to be a goal node by a processor, a message can be broadcast to inform all processors to terminate. If a speculative node is found to be a goal node, this node is simply added to *RL* because it may not be an optimal goal node.

*PLA** can terminate, failing to reach a goal node, when $WL^i = \emptyset$. $WL^i = \emptyset$ if and only if $WL_i{}^i = \emptyset$, for all *j* = 0.... P— 1. This state can be recognized and broadcast to all processors at the end of the node transfer procedure.

2.8 Summary

The *PIA** algorithm can be summarized below:

$WL = \{s\}; RL = \emptyset;$

loop *until a goal node is reached or WL is empty*
  { Start node expansion procedure }
  for *each processor]*
    expand all mandatory nodes from WL_j;, *and add successors to RL using the multiplex round-robin successor distribution algorithm:*
    while *there is a processor expanding a mandatory node*
      expand the best speculative node from WL_j *and add successors to RL using the multiplex round-robin successor distribution algorithm;*
  { Start node transfer procedure }
  for *each processor j*
    insert all nodes from RLj to \\'Lj:

Mandatory nodes and speculative nodes are discriminated by comparing their cost to a threshold t. A node n is a mandatory node if f(n) < t; otherwise, it is a speculative node. Initially, / — *h(s).* Successive / values are computed during the node expansion procedure by (Fq I) presented in Section 2.3.

3 Admissibility

It is well known that if the heuristic function $h \leq h^*$ (the shortest distance to a goal node), then A^* is admissible. PIA^* is also admissible under the same conditions.

PIA^* has a similar property found in A^* [Nilsson, 1980]; i.e.:

Lemma 1 *$\forall i$ before termination, $\exists n' \in WL^i$ such that n' is on an optimal path from s to a goal node and $f(n') \leq f^*(s)$.*

Proof: (By induction on the number of iterations) Initially, $WL^0 = \{s\}$ and $f(s) = h(s) \leq h^*(s) = f^*(s)$. Hence, s is the node n'. Assume the lemma is true for all WL^i, $i \leq k$, and n^k is a node, in WL^k, which is on an optimal path and $f(n^k) \leq f^*(s)$. If node n^k was not expanded at iteration k, then node n^k is in WL^{k+1} and the lemma is proved. If n^k was expanded at iteration k (n^k was not a goal node; otherwise PIA^* would have terminated.), then, since n^k is on an optimal path, one of its successors n' is on an optimal path. From (Eq2), node n' is in WL^{k+1}, and

$$f(n') = g(n') + h(n') = g^*(n') + h(n') \leq f^*(s).$$

(Q.E.D.)

From (Eq1), we know $t^{i+1} \geq t^i$, for every i. Moreover, $t^{i+1} = t^i$ if and only if there is a node $n \in Suc(M^i)$ with $f(n) \leq t^i$ (Note that this node n will be expanded at iteration $i+1$). Because we normally assume the arc cost in the state-space graph is positive, this situation cannot happen forever. Thus:

Lemma 2 *$\forall i\ t^{i+1} \geq t^i$ and $\exists m$ such that $t^{i+m} > t^i$.*

To show PIA^* is admissible, we need to prove another important property of PIA^*.

Lemma 3 *$\forall i$ before termination, $t^i \leq f^*(s)$.*

Proof: (By induction on the number of iterations) Initially, $t^0 = f(s) \leq f^*(s)$. Assume $t^i \leq f^*(s)$, for all $i \leq k$. We know $t^{k+1} \geq t^k$. If $t^{k+1} = t^k$, then $t^{k+1} \leq f^*(s)$ by the induction hypothesis and the lemma is proved. If $t^{k+1} > t^k$, then, from (Eq1), $t^{k+1} = min\{f(n) \mid n \in Suc(M^k) \cup S^k\}$. By Lemma 1, the set $M^k \cup S^k = WL^k$ has a node n with $f(n) \leq f^*(s)$. It is straightforward to see that this is also true for the set $\Sigma = Suc(M^k) \cup S^k$ (see the proof of Lemma 1); i.e., there is at least one node $n \in \Sigma$ such that $f(n) \leq f^*(s)$. Therefore, $t^{k+1} \leq f^*(s)$.

(Q.E.D.)

By Lemma 2, the threshold will eventually approach $f^*(s)$, if $f^*(s)$ is finite. By Lemma 3, the threshold will never exceed $f^*(s)$ before termination. Therefore, PIA^* will terminate by finding an optimal path if there is one. That is, we have proved:

Theorem 1 *Algorithm PIA^* is admissible.*

4 Comparison of PIA^* to A^*

Some interesting relationships exist between PIA^* and A^*. It is well known that any node n with $f(n) < f^*(s)$ will eventually be selected for expansion by A^*. A node n with $f(n) = f^*(s)$ may or may not be expanded by A^* depending on when the first goal node is reached. However, in the worst case A^* will expand all nodes n with $f(n) \leq f^*(s)$. We have shown that $t^i \leq f^*(s)$. Because a mandatory node n in PIA^* has $f(n) \leq t^i$, $f(n) \leq f^*(s)$, for all $n \in M^i$. That is:

Observation 1 *All mandatory nodes in PIA^* will be expanded by A^* and PIA^* in the worst case.*

In practice, PIA^* may expand more nodes or fewer nodes than A^*. This phenomenon is similar to the *parallel search anomalies* discovered previously [Lai and Sahni, 1984][Quinn and Deo, 1986]. These anomalies can become noticeable when there are many nodes with cost equal to $f^*(s)$.

Interestingly, at the last iteration of PIA^*, some mandatory nodes n, $f(n) < f^*(s)$, which would be always expanded by A^*, may not be expanded by PIA^* when PIA^* terminates with an optimal solution path. This *benign* anomaly of PIA^* is analogous to the finding by Mero [Meor, 1984] that the costs of ancestors can be used to construct a more informed cost function that will always dominate A^*'s cost function.

When only one processor is used to run *PIA**, no speculative nodes would he expanded at each iteration, and obviously processor synchronization is not required. *PIA** then proceeds very similarly to *A** except that the order of the node expansion may he different because a successor generated by *PIA** is not immediately considered for expansion. However, from Observation I, we know:

Observation 2 *PIA* using a single processor perforins 1he saint number of node expansions as A* in the worst case.*

That is, *PIA** using one processor can perform as well as.A*.

5 Analysis and Experimental Results

In this section, we will give an informal analysis of the load balancing, scalability, and speedup of *PIA**. Experimental results will be also shown to support the analysis.

To empirically test the *PIA** algorithm, a three-machine flow-shop scheduling problem was implemented on a BBN Butterfly Multicomputer. The three-machine flow-shop scheduling problem is to schedule a given set of jobs on three machine's such that the span of time to finish all of the jobs is minimized. The lower bound function described by Ignall and Schrage [1965] was used as the cost function f. We ran a 12 job problem using 5,, 10, 20, :30, 10, 50, (50, 70 and SO processors on a Butterfly and compared their results. We could not run this problem on a single processor because of insufficient memory available on one Butterfly processor. The experimental results are summarized in Table I. The detailed experimental results can be found in [Huang and Davis, 1981)].

Table 1: Summary of PIA^* Experimental Results

#Proc	#Nodes		
	Mandatory	Speculative	Total"
5	13242	1162	14404
10	10376	2417	12793
20	10086	3920	14006
30	7132	4067	11199
40	9481	4647	14128
50	7856	5304	13160
60	6378	5553	11931
70	6151	7661	13812
80	6058	7470	13528

#Proc	#Itn	Time(sec)	Speedup	Efficiency
5	35	48.81	5.0"	1"
10	31	21.47	11.4	1.14
20	28	11.83	20.6	1.03
30	25	6.59	37.0	1.23
40	22	5.97	40.9	1.02
50	22	4.77	51.2	1.02
60	23	3.84	63.6	1.06
70	25	4.06	60.1	0.86
80	23	3.69	66.1	0.83

"Mandatory nodes plus speculative nodes
"Assumed

5.1 Load Balancing

Load balancing is one of the main factors determining the efficiency of a parallel algorithm. Processors not only have to be kept busy but also have to be busy on productive work a high percentage of the time in order to

attain good load balancing and to obtain almost linear speedup.

One of the main features that contribute to the load balancing of the *PIA** algorithm is its *symmetric structure.* Each processor in *PIA** maintains the same type of data structures, executes the same type of operations, and interacts with other processors in the same environment. By symmetry principle, probabilistically, the nodes tend to distribute evenly both in number and in cost among processors. In Figure 2, we plot, from our experiments, the total number of nodes expanded at each processor, and the number of nodes left in the work list at each processor when 70 processors were used. The distribution is very even among processors.

At each iteration of *PIA*,* there are two synchronized procedures: the node expansion procedure and the node transfer procedure. We will study the load balancing of these two procedures respectively.

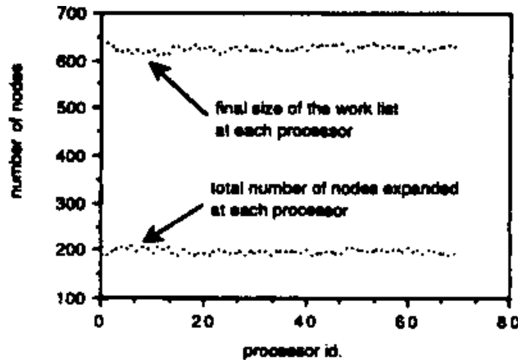

Figure 2: Load Distribution in 70 Processors

5.1.1 Load Balancing in the Node Expansion Procedure*

Processors are kept almost constantly busy in the node expansion procedure because the speculative computation is performed additionally at each processor to use the waiting time for processor synchronization. This arrangement has been very effective in our experiments. As we can see from Figure 2, the total number of nodes expanded was nearly constant at each of the 70 processors.

Rut, how productive is the speculative computation at each iteration? If MP is evenly distributed among all processors, then very few speculative nodes will be selected for expansion. We have argued above[1] that this is generally so probabilistically, owing to the symmetric structure of the *PIA** algorithm. A speculative[1] node selected for expansion by a processor is the best node available to this processor's work list at that time. At each iteration, any speculative node *n* with *f(v)* < f*(*) can become a mandatory node at one of the subsequent, iterations, and will eventually be expanded by *A** as well. Hence, the speculative computation which expands this type of speculative nodes is also productive.

In our experiments (see Table I), the total number of speculative nodes expanded tended to increase when more processors were used; but the total number of nodes (mandatory nodes and speculative nodes) expanded did not tend to increase (The zigzagging is attributed to the parallel search anomalies). Furthermore, the total number of iterations tended to decrease as more processors were used (see Table I). These results empirically show that most of the speculative computation performed was productive in our experiments.

5.1.2 Load balancing in the Node Transfer Procedure

If each WL_j, $j = 0 \cdots P-1$, is maintained as a *heap* [Horowitz and Sahni, 1978], then the time taken for a processor j to finish the node transfer procedure is proportional to $\overline{RL_j} \log(\overline{WL_j} + \overline{RL_j})$, where $\overline{RL_j}$ is the number of nodes in RL_j and $\overline{WL_j}$ is the number of nodes in WL_j before the node transfer procedure is executed.

Therefore, the load balancing of the node transfer procedure depends on how uniform the $\overline{RL_j}$'s and the $\overline{WL_j}$'s are.

Observation 3 $|\overline{RL_i} - \overline{RL_j}| \leq P$, $i \neq j$.

Proof: By the multiplex round-robin algorithm, the numbers of successors received by RL_i and RL_j from a processor can differ at most by 1. There are P processors; hence, $|\overline{RL_i} - \overline{RL_j}| \leq P$, $i \neq j$.

(Q.E.D.)

The multiplex round-robin algorithm for the successor distribution has a deterministic worst case which is independent of the problem size but can grow linearly with the number of processors. The probability of the worse case occurring is very low as the number of processors increases. In fact, when the a priori characteristics of a problem instance are unavailable, because of the symmetry, the expected maximum difference of *RLj*'s* should be close to zero.

Let us consider Figure 2 again. Recause the total number of nodes expanded and the final size of the work list at each processor are very uniform, we expect that the size of *RLj's* before the node transfer procedure should be approximately uniform at each iteration. Note that each processor at each iteration only expanded less than 10 nodes on the average (about 200 nodes expanded in over 20 iterations) to obtain this load balancing.

5.2 Scalability

Computational requirements for most interesting combinatorial search problems grow very quickly with problem size. It is not difficult to find problems which can use millions of processors for the PIA* algorithm, ('an the *PI A** algorithm scale accordingly provided that the underlying multicomputer is scalable? Fxcept for the successor distribution and the processor synchronization, operations in *PI A** are completely local. Ry using *tlu butltrflu barrier* suggested by Brook III [1080], processor synchronization can scale very well with increased numbers of processors.

Successor distribution is accomplished by the help of a reception list at each processor. Since reception lists are accessed by all processors, a legitimate concern would be whether contention for them would lead to significant degradation or a bottleneck. An equal number of reception lists and processors exists, and each processor has equal probability of placing a node on any reception list. Recause if is a simple list structure, a reception list can

be protected with a very short critical region. We expect that contention for the reception lists l>y processors will not be a problem in practice and its seriousness does not increase with the number of processors. In [Huang and Davis, 1989], we use elementary queueing theory to support this expectation.

5.3 Speedup

Since *P1A** proceeds iteratively, we will analyze the speedup in a single iteration to project, the overall speedup. Speedup for *P* processors is normally defined as the ratio of execution time using one processor and that using *P* processors; i.e.,

$$Sp(P) = \frac{T_1}{T_P}.$$

Let us define 11', = T_1 and $W_P = P \cdot T_P$ as the total amount of work required when using one processor and *P* processors respectively. Assume that the same set of nodes are expanded in W,, and W_P,. Then, to compare W_s, and lip, we note that a fraction of W,, f IV,, 0 < / < 1, is the work to add successors to *RL*. When *P* > I processors are used, the same amount of work is converted to nonlocal successor distribution operations. Provided that an assumed scalable multicomputer is used to run *PLA**, is converted to f.H, • $c_1 \log P$ in H_P, where c_1 is a constant which depends on the efficiency of the communication network in the underlying multicomputer. In addition, \Yp also contains the computation cost for processor synchronization. Assuming t*he butterfly barrier* is used, each processor needs to run log *P* stages and in each stage communicates with an other processor. The total cost can then be expressed as $C_2 P \log^2 P$, where c_2 is another constant. Therefore, we can express H'p as:

$$W_P = (1-f)W_s + fW_s c_1 \log P + c_2 P \log^2 P.$$

and the speedup is:

$$Sp(P) = \frac{P}{(1-f) + fc_1 \log P + \frac{c_2 P \log^2 P}{W_s}}$$

The above equation shows how the communication overhead introduced by successor distribution operations, and the processor synchronization overhead would affect the speedup of *PIA**. Since only simple operations are required for successor distribution, the fraction f should be very small for most heuristic search problems. The significance of processor synchronization overhead is inversely proportional to $\frac{W_s}{P}$; i.e., the synchronization overhead is less significant if each processor expands more nodes at each iteration. For exponential search problems, we argue that there are normally many mandatory nodes at each iteration. Consider the 20-city Traveling Salesman Problem as an example. The size of the search state space is 20!. Assuming a 32-bit integer is used to represent the cost of a state, there are at most 2^{32} different- cost values. Then, on the average, there are over 10^8 states assigned the same cost value! The speedup, based on the execution time taken when 5 processors were used, is shown in Table 1. On the average, fewer than 10 nodes were expanded at each iteration when more than GO processors were used (see [Huang and Davis, 1989]). Overall, only about 14000 nodes were expanded in total in every experiment. The problem size was chosen so that the benchmarking time was reasonable to measure, and it was just barely enough to effectively utilize 80 processors. Prom our analysis, *PIA**, similar to most other parallel algorithms, will perform better for problems with larger problem sizes.

As a comparison, we implemented the same problem using the central queue approach on the same machine. The timing and speedup results are summarized in Table 2. The maximum speedup was less than 3 and the execution time had no sign of improvement when we increased the number of processors to 50.

Because of memory limitations, we were unable to obtain the speedup of *PLA** based on *A** running on a single processor of Butterfly in Table 1; however, a small problem was tested to compare actual running times of *PLA** and *A**. The result is shown in Table 3. The speedup of *PLA** in Table 3 is based on *A**. In our implementations, the run time performance of *PLA** using one processor is very close to *A** if both algorithms expand the same number of nodes. The total number of nodes expanded depends on when a goal node is reached, and it can vary widely if there are many nodes with cost equal to the optimal solution cost. Also from Table 3, the parallelization overhead of *P1A** is not significant for this small problem size. (Note that when more than one processor are used, remote memory reference has been reported to be about 5 times more expensive than local memory reference on Butterfly.)

Table 2: Experimental Results for the Central Queue Approach ((Cf. Table 1)

#Proc	Time(sec.)	SpeedUp	Efficiency
1	300.1	1	1
2	154.0	1.95	0.98
3	116.0	2.59	0.86
4	111.8	2.68	0.67
5	108.0	2.78	0.56
10	114.2	2.63	0.26
20	109.4	2.74	0.14
30	109.2	2.75	0.09
40	111.9	2.68	0.07
50	117.7	2.55	0.05

Table 3: Performance Comparison of *PIA** and *A**

*A**

#Nodes Expanded	Time(sec.)
582	3.64

*PIA**

#Proc	#Nodes Expanded	Time(sec.)	Speedup"
1	775	4.67	0.78
2	635	2.10	1.73
3	561	1.25	2.91
4	554	0.96	3.79
5	591	0.88	4.14

"Compared to *A**

6 Concluding Remarks and Future Research

We have presented a distributed best-first heuristic search algorithm, *PLA**. We proved the algorithm is admissible and gave an informal analysis of its load bal-

ancing, scalability and speedup. A (low-shop scheduling problem was chosen to implement the PLA* algorithm on the BBN Butterfly Multicomputer using up to SO processors. The experimental results were encouraging. It seems that this algorithm can achieve almost linear speedup on a large number of processors with a relatively small problem size. We expect this algorithm can be efficiently implemented on a large class of scalable mult icomputers and can solve a variety of combinatorial optimization problems. However, because PIA* has not been extensively tested on many types of problems and mult icomputers, its actual limitations and advantages have yet to be* more carefully evaluated in future tests.

The PLA* algorithm uses roughly the same amount of memory as A*. A* often fails to solve an exponential search problem because it runs out of space very quickly. Although PLA* can effectively use the combined memory of a loosely-coupled multicomputer, it is also vulnerable to the memory shortage problem when trying to solve large exponential search problems. We are developing a linear space variant of PIA* and investigating an implementation on the Connection Machine, further results will be published in a sequel to this paper.

7 Acknowledgements

The authors would like to deeply thank Richard Korf who reviewed and critiqued early drafts of this paper. Vipin Kumar's comments on our early versions of PI A* have been very valuable to us. Special thanks to Perry Thorndyke and N. S. Sridharan who encouraged and supported this research.

References

[Anderson and Then, 1987] Anderson, S. and Then, M. C, Parallel branch-and-bound algorithms on the hypereube, in Hypercube Multiprocessors 1987, M. 1. Heath, ed SIAM Press, Philadelphia, PA, 1987.

[BBN, 1985] BBN Laboratories, Butterfly Paralld Proce ssor Overview, Holt, Heranek and Newman, Cambridge, Massachusetts, Dec. 1985.

[Hrook III, 1980] Hrook III, E.D., The butterfly barrier International Journal of Parallel Programming, vol. 15, no. I, August 1986.

[Mart et al., 1908] Mart, PL., Nilsson, N.J. and Raphael, H., .4 formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Systems, Science and Cybernetics, SSC-42, no. 2, 1968, pp. 100 107.

[Hillis, 1985] Hillis. D., The Connection Machine, Cambridge, MA, MIT Press, 1985.

[Horowitz and Sahni, 1978] Horowitz, 10. and Sahni, S., Fundamentals of Compute r Algorithms, Rockville, MI), Computer Science Press, 1978.

[Huang and Davis, 1987] Huang, S. and Davis L. S., .4 tight uppe r-bound for the speedup of paralld best-first branch-and-bound algorithms, Technical Report, CS-TR-1852, Computer Science Department, University of Maryland, College Park, April 1987.

[Huang and Davis, 1989] Huang, S. and Davis L S., PLA*: an admissible distribute d he urtstic searrh algorithm, Technical Report, CS-1 R-2233 Computer Science Department, University of Marvland, College Park, April 1989.

[ignall and Schrage, 1965] Ignall, E. and Schrage. L., Application of the branch and bound technique to some flow-shop sche duling problems. Opus. Res., 13, no 3, 1965, pp 100-112.

[Intel, 1980] intel PSC System Overview, Intel Scientific Computers, 1980.

[Kleinrock, 1975] Kleinrock, L., Queueing Systems: Theory, Vol 7, New York: Wiley, 1975.

[Kumar et al, 1988] Kumar, V., Ramesh K. and Rao, V. N., Paralld heuristic search of state-space graphs : a summary of results, in Proceedings of the 1988 National Conference on Artificial Intelligence, August 1988.

[Lai and Sahni, 108 I] Lai, T. II. and Sahni, S., Anomalies in parallel branch-and-bemnd algorithms, Comm. ACM, vol 27, 198 I, pp. 59-1-002.

[Mero, 1984] Mero, L , A heuristic search algorithm with modifiable estimate, Artificial Intelligence 23, 1, May 1984, pp. 13-27.

[Mohan, 1982] Mohan, .)., .4 study in parallel computation - the Pwveling Salesman Problem, Technical Report CMP-('S-82-130, ('omputer Science Department, Carnegie-Mellon University, August, 1982.

[Nilsson, 1980] Nilsson, N. J., Principles of Artificial Intelligence, Palo Alto, CA, Tioga Press, 1980.

[Paige and Kruscal, 1985] Paige, R. C. and Kruskal, C. P., Parallel algorithms for shortest path problems, in Proc hit I. Conf. on Parallel Processing, August 1985, pp. 14-20.

[Pearl, 1984] Pearl, J., Heuristics, Addison-Wesley, Reading, MA, 1984.

[Quinn and Deo, 1980] Quinn, M. J. and Deo, N., An upper bound for the speedup of parallel branch-and-bound algorithms, HIT, vol 0, no. I, March 1980).

[Quinn, 1987] Quinn, M. J., Imple menting best-first branch-and-bound algorithms on hypercubc multicomputers, in Hypercuhe Multiprocessors 1987, M. T. Heath, ed., SIAM Press, Philadelphia, PA, 1987.

[Rao and Kumar, 1988] Rao, Y. N. and Kumar, Y , Concurrent access of priority que ues, ILLP Trans, on Computers, vol 37, no. 12, Dec. 1988, pp. 1057-1005.

[Wall and Lva Ma. 1984] Wall, H. \Y. and Kva Ma, Y. \\ ., MAXIP- A multicompute r architecture for solving combinatorial e rtre mum-se arch problems, IEEE, Trans, on Computers, vol 33, May 1984, pp. 377-390.