Quest Journals
Journal of Software Engineering and Simulation
Volume 9 ~ Issue 3 (2023) pp: 38-49
ISSN(Online) :2321-3795 ISSN (Print):2321-3809
www.questjournals.org

Research Paper

An Efficient Load Balancing Algorithm for Parallel Searching and Synchronization

[1]Bennett, E. O. and[2]Igiri, C. G.
Department of Computer Science, Rivers State University, Port Harcourt, Nigeria

Abstract
In parallel tree search environments, it is likely that some nodes are heavily loaded while others are lightly loaded or even idle. It is desirable that the workload is fully distributed among all nodes to utilize the processing time and optimize the whole performance. A load balancing mechanism decides where to migrate a process and when. This paper introduces a load balancing mechanism as a novel scheme to support the reliability and to increase the overall throughput in parallel tree search environments. To alleviate the effects of processor idleness and non-essential work, a dynamic load balancing scheme called Preemptive Polling Scheme (PPS) is proposed. This scheme employs a near-neighbour quantitative load balancing method by detecting and correcting load imbalance before they occur during the search. Message Passing Interface (MPI) parallel communication model was used for interprocessor communication. The search algorithm was employed to solve a Discrete Optimization Problem (DOP)(Knapsack problem) on the multi-processor system. The results show that PPS brings about scalability and efficiency in the system, as there is time reduction in the execution of the parallel tree search algorithm over its sequential counterpart is solving the DOP.

Keywords: *Parallel tree search, dynamic load balancing, distributed systems, algorithms*

Received 10 Mar., 2023; Revised 20 Mar., 2023; Accepted 23 Mar., 2023 © The author(s) 2023. Published with open access at www.questjournals.org

I. Introduction

Very often applications need more computing power than a sequential computer can provide. One way of overcoming this limitation is to improve the operating speed of processors and other components so that they can offer the power required by computationally intensive applications. Even though this is currently possible to certain extent, future improvements are constrained by the speed of light, thermodynamic laws, and the high financial cost of processor fabrication. A viable and cost-effective alternative solution is to connect multiple processors together and coordinate their computational efforts. The resulting systems are popularly known as parallel computers, and they allow the sharing of computational task among multiple processors.

As [1] points out; there are three ways to improve performance:

i. Work harder,
ii. Work smarter, and
iii. Get help

In terms of computing technologies, the analogy to this mantra is that working harder is like using faster hardware (high performance processors or peripheral devices). Working smarter concerns doing things more efficiently and this revolves around the algorithms and techniques used to solve computational tasks. Finally, getting help refers to using multiple computers to solve a computational task.

An important component of many scientific applications is the assignment of computational loads onto a set of processors. The problem of finding a task to processor mapping that minimizes the total execution time is known as MAPPING problem [2]. The amount of processing time needed to execute all processes assigned to a processor is called workload of a processor [3]. Load balancing strategies are used to tackle the inefficiencies of starvation and non-essential work (and hence that of memory overhead). Load balancing involves the distribution of jobs throughout a networked computer system, thus increasing throughput without having to obtain additional or faster computer hardware. Load balancing is to ensure that every processor in the system does approximately the same amount of work at any point in time [15]. An important problem here is to decide how to achieve a balance in the load distribution between processors so that the computation is completed in the shortest possible time. Since the subject of this paper is load balancing approaches, we will confine our attention to tree search spaces.

The departure from strict global-rank ordered search and a distributed memory implementation introduce several inefficiencies in parallel search algorithms:

Starvation: This is defined as the total time (over all processors) spent idling and occurs when processors run out of work.

Non-essential work: This is the total time spent processing non-essential nodes. It arises because processors perform local-rank ordered search (processors expand nodes)

i. ***Memory Overhead:*** This is caused by the generation and storage of non-essential nodes. Therefore, tackling non-essential work automatically takes care of the memory overhead problem.

ii. ***Duplicated work:*** This is the total extra time associated with pursuing duplicate search spaces and is due to inter-processor duplicates, i.e. duplicate nodes that arise in different processors, when the search space is a graph.

iii. The above inefficiencies grow with the number of processors ***P*** used thus causing the efficiency $E = T1/(P.T_P)$ to deteriorate; here ***T1*** denotes the sequential execution time and represents the essential work for the problem in terms of the amount of time spent processing essential nodes, and ***Tp*** denotes the execution time on Processors. We use work density to refer to the ratio ***T1/P***.

II. Literature Review

2.1 Application of Tree Search Algorithms

In areas like discrete optimization, artificial intelligence, and constraint programming, tree search algorithms are among the key techniques used to solve real-world problems. This section contains several examples of successful application of tree search algorithms. For instance, [5] developed a parallel game tree search algorithm called ZUGZWANG that was the first parallel game tree search software successfully used to play chess on a massively parallel computer and was vice world champion at the computer chess championships in 1992. Applegate, Bixby, Chvatal, and Cook used a parallel implementation of branch and cut, a type of tree search algorithm [6], to solve a Traveling Salesman Problem (TSP) instance with 85,900 cities in 2006. The number of variables in the standard formulation of TSP is approximately the square of the number of cities. Thus, this instance has roughly 7 billion variables.

With the rapid advancement of computational technologies and new algorithms, one of the main challenges faced by researchers is the effort required to write efficient software. To effectively handle discrete optimization problems, there is need to incorporate problem-dependent methods (most notably for dynamic generation of variables and valid inequalities) that typically require the time-consuming development of custom implementations. It is not uncommon today to find parallel computers with hundreds or even thousands of processors. Scalability is still not easy to obtain in many applications, however, as computing environment have become increasingly distributed, developing scalable parallel algorithms has become more and more difficult.

In the following sections, we introduce some definitions and background related to tree search, parallel computing, and parallel tree search. Also, we review previous worksin parallel tree search.

2.2 Definitions

This section presents the necessary background on tree search. Here, we introduce the definitions and notations relevant for describing general tree search problems and algorithms [7, 8, 9].

Definition 2.1: A graph **or** undirected graph G = (N, E) consists of a set of **N** of vertices or nodes and a set **E** of edges whose elements are pairs of distinct nodes.

Definition 2.2: A walk is an alternating sequence of nodes and edges, with each edge being incident to the nodes immediately preceding and succeeding it in the sequence.

Definition 2.3: A path is a walk without any repetition of nodes.

Definition 2.4: Two nodes ***i*** and ***j*** are connected if there is at least one walk from node ***i***to
node***j***.

Definition 2.5: A connected graph is a graph in which every pair of nodes is connected.

Definition 2.6: A tree is a connected graph that contains no cycle.

Definition 2.7: A subtree is a connected subgraph of a tree.

Definition 2.8: A problem is a set of decisions to be made subject to specified constraints and objectives.

Definition 2.9: A state describes the status of the decisions to be made in a problem. e.g., which ones have been fixed or had their options narrowed and which ones are still open.

Definition 2.11: A successor function is used to change states, i.e., fix decisions or narrow the set of possibilities. Given a particular state ***x***, a successor function returns a set of < action, successor > ordered pairs, where each action is one of the possible activities in state ***x*** and each successor is a state that can be reached from state ***x*** by applying the action.

Definition 2.12: A state space of a problem is the set of all states reachable from the initial state. The initial state and successor functions implicitly define the state space of a problem.

Definition 2.13: A path in a state space is a sequence of states connected by a sequence of actions.

Definition 2.14: A goal test function determines whether a given state is a goal state.

Definition 2.15: A path cost function assigns a numeric cost to each path.

Definition 2.16: A solution to a problem is a sequence of actions that map the initial state to a goal state. A solution may have a path cost, which is the numeric cost of the path in the state space generated by the sequence of actions applied to reach that state.

Definition 2.17: An optimal solution is a solution that has lowest path cost among all solutions.

Definition 2.18: The feasible region of a problem is the set of all solutions.

Definition 2.20: Expanding or branching is the process of applying a successor function to a state to generate a new set of states.

Definition 2.20: Processing is the procedure of computing the path cost of a state, checking if the state is a goal state, and expanding the state.

Definition 2.21: Tree search is the process of finding a solution or optimal solution that map an initial state to a goal state. Tree search involve the iterative steps of choosing, processing, and expanding states until either there are no more states to be expanded or certain termination criteria are satisfied. Tree search can be divided into two categories: feasibility search that aims at finding a feasible solution and optimality search that aims at finding an optimal solution.

A node is a bookkeeping data structure used to represent a state, while a state corresponds to a representation of the status of the decisions to be made. Two different nodes can contain the same state if that state can be generated via two different paths, although this is to be avoided if at all possible.

A node's description generally has the following components [10]:

i. **State***:* The state in the state space to which the node corresponds;
ii. **Parent***:* The node in the search tree that generated this node;
iii. **Action***:* The action that was applied to the parent to generate the node;
iv. **Path cost***:* The cost of the path from the root to the node; and
v. **Depth***:* The number of steps along the path from the root.

A variety of algorithms have been proposed and developed for tree search. Tree search algorithms are among the most important search techniques to handle difficult real-world problems. Due to their special structure, tree search algorithms can be naturally parallelized, which makes them a very attractive area of research in parallel computing. The main elements of tree search algorithms include the following:

i. **Processing method***:* A method for computing path cost and testing whether the current node contains a goal state.
ii. **Successor function***:* A method for creating a new set of nodes from a given node by expanding the state defined by that node.
iii. **Search strategy***:* A method for determining which node should be processed next.
iv. **Pruning rule***:* A method for determining when it is possible to discard nodes whose successors cannot produce solutions better than those found already or who cannot produce a solution at all.

To implement tree search, we simply keep the current list of leaf nodes of the search tree in a set from which the search strategy selects the next node to be expanded. This is conceptually straightforward, but it can be computationally expensive if one must consider each node in turn in order to choose the "best" one. Therefore, each node is usually assigned a numeric priority and stores the nodes in a priority queue that can be updated dynamically.

There are several reasons for the impressive progress in the scale of problems that can be solved by tree search. The first is the dramatic increase in availability of computing power over the last two decades, both in terms of processor speed and memory size. The second is significant advancements in theory. New theoretical research has boosted the development of faster algorithms, and many techniques once declared impractical have been "re-discovered" as faster computers have made efficient implementation possible. The third is the use of parallel computing. Developing a parallel program is not as difficult today as it once was. Several tools, like OpenMP [11], Message Passing Interface (MPI) [12], [13], and Parallel Virtual Machine (PVM)[14],provide users a handy way to write parallel programs. Furthermore, the use of parallel computing has become very popular. Many desktop PCs now have multiple processors, and affordable parallel computer systems like Beowulf clusters appear to be an excellent alternative to expensive supercomputers.

Load Balancing Strategies

Load balancing strategies can have three categories based on initiation of processes:

i. **Sender Initiated:** In this type of strategy, load transfer is initiated by the sender (donour) processor.
ii. **Receiver Initiated:** In this type of strategy, load transfer is initiated by the receiver (sink) processor.
iii. **Symmetric:** This is a combination of both sender and receiver initiated.

Depending on the current system state, load balancing strategies can be divided into two categories as **static** and **dynamic** load balancing.

Static Load Balancing

In these types of algorithms, no dynamic load information is used, the assignments of the tasks to the processors are made *a priori* using task information (arrival time, average execution time, amount of resources needed, and their inter-process communication requirements), or probabilistically. However, in reality this whole information may not be known a priori (at compile time) and this will get worse if we work in heterogeneous systems where task execution times on processing nodes will vary according to capacity of hosts.

Dynamic Load Balancing

The main problem with the **Static** load balancing is that they assume too much task information which may not be known in advance even if it is available; intensive computation may be involved in obtaining the optimal schedule. Because of this drawback much of the interest in load balancing research has shifted to dynamic load balancing that considers the current load conditions (execution time) in making task transfer decisions. So here the workload is not assigned statically to the processor, instead, this workload can be redistributed amongst processors at the runtime as the circumstances changes i.e., transferring the tasks from heavily loaded processors to lightly loaded ones.

Dynamic load balancing continually monitors the load on all participating processors, and when the load imbalance reaches some predefined level, this redistribution of workload takes place. But as this monitoring steals CPU cycles, care must be taken as to when it should be invoked. This redistribution does incur extra overhead at execution time.

Quantitative Load Balancing

In this approach, each processor monitors its ***activelen***(active nodes) periodically and reports any significant changes in it to its neighbours. Also, each processor assumes that the processor space comprises its neighbour and itself only.

Load balancing strategies are used to tackle the inefficiencies of starvation and non-essential work (and hence that of memory overhead), while duplicate pruning strategies [are required to minimize duplicated work.

Since the subject of this paper is load balancing, we will confine our attention to tree search spaces.

Load Balancing Strategies

Several methods have been proposed to achieve quantitative load balance [15, 16, 17]. Here, we critically analyse two representative schemes namely, Neighbourhood Averaging (NA) and Round-Robin . In the Round-Robin (RR) strategy, a processor that runs out of nodes requests work from its busy neighbours in a round-robin fashion, until it is successful in procuring work. The donor processor grants a fixed fraction of its active nodes to the acceptor processor. The drawback of this scheme is that a number of decisions such as the next processor to request from and the fraction of work that should be granted are oblivious to the load distribution in the neighbouring processor.

The Neighbourhood Averaging (NA) strategy tries to achieve quantitative load balance by balancing the number of active nodes (active-len) among neighbouring processors. For this purpose, each processor reports the current active-len value to its neighbours when it has changed by some constant absolute amount *delta*. There are two main drawbacks in this scheme. First, work transfer decisions rely solely on the load distribution around the source processor, and not that around the sink processor as well. This can give rise to two types of problems. Firstly, since this is a source-initiated strategy, it can happen that multiple source neighbours may simultaneously attempt to satisfy the deficiencies of a sink processor, thus in all likelihood converting the latter to a “source” relative to its previously source neighbours. As a result, thrashing of work will occur. Further, it is also possible for work transfer decisions to be contrary to the goal of good load balance. The second major drawback in this strategy is that load information is disseminated when absolute changes in load occur rather than percentage changes---small load changes are more important at lower loads than at heavier loads. By not taking this into account, load reports may either be too frequent (high communication overhead) or too widely spaced (poor load balancing decisions).

Isoefficiency Analysis Review

Isoefficiency analysis[18] is also used in characterizing the scalability of parallel systems. The key in isoefficiency analysis is to determine an isoefficiency functionthat describes the required increase of the problem size to achieve a constant efficiency with increasing computer resources. A slowly increasing isoefficiency function implies that small increments in the problem size are sufficient to use an increasing number of processors efficiently; thus, the system is highly scalable. The isoefficiency function does not exist for some parallel systems if their efficiency cannot be kept constant as the number of processors increases, no matter how large the problem size increases. Using isoefficiency analysis, one can test the performance of a parallel program on a few processors, and then predict its performance on a larger number of processors. It also helps to study system behavior when some hardware parameters, such as the speeds of processor and communication, change. Isoefficiency analysis is more realistic in characterizing scalability than other measures like efficiency, although it is not always easy to perform the analysis.

3.System Design

In figure 1, the architectural design of the Preemptive Polling Schemeis presented. The “finished available work” section in a generic load balancing architecture is replaced in figure 1 with the “PPS”.

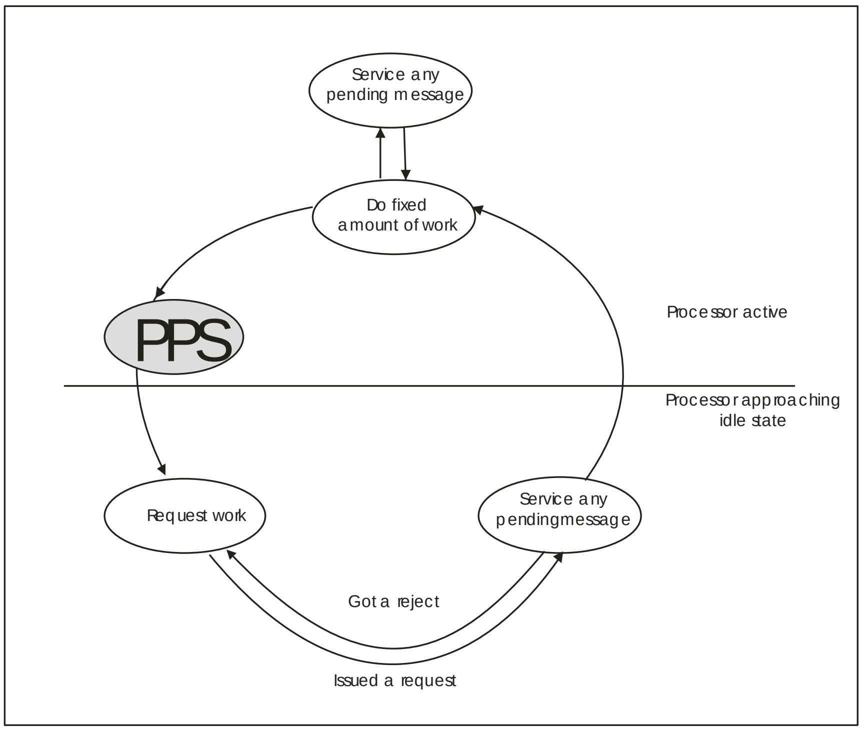
Service any pending message
Do fixed amount of work
PPS
Processor active
Processor approaching idle state
Request work
Service any pendingmessage
Got a reject
Issued a request

Fig. 1: Preemptive Quantitative Load Balancing Architecture

The Preemptive Polling Scheme(PPS)adopts a quantitative load balancing method to reduce idleness (idle time) and load balancing overhead. It is organized as a four-phase process: (i) processor load evaluation, (ii) load balancing profitability, (iii) task migration, and (iv) task selection/transfer.

The first and fourth phases of the model are application dependent and purely distributed. Both phases can be executed independently on each individual processor.Hence, theProcessor Load EvaluationPhaseisreduced to a simple count of the number of tasks pending execution. Similarly, the task selection strategy is simplified since no distinction is made between tasks.

For the case where tasks are created dynamically, if the arrival rate is predictable then this information can be incorporated into the load evaluation, if not predictable, then the potential arrival of new tasks can effectively be ignored. As the program execution evolves, the inaccuracy of the task requirement estimates leads to unbalanced load distributions. The imbalance must be detected and measured (phase ii) and an appropriate migration strategy devised to correct the imbalance (phase iii). These two phases may be performed in either distributed or centralized fashion.

In our scheme, each processor monitors its (active nodes) “active-lens” periodically and reports any significant changes in it to its neighbours. In our implementation, a 10% change is reported to the neighbours. Also, each processor assumes that the processor space comprises its neighbours and itself only.

Let ***wi***= Amount of work in terms of active nodes available at processor ***i***
Wavg,,j = Average amount of work per processor available with ***i*** and its neighbours
Let ***δji = wj – Wavg,j***= surplus amount of work at processor ***j***with respect to processor **Wavg,j**

To achieve quantitative load balance between ***i*** and its neighbours, each processor should have **Wavg,j** amount of work. This means that each neighbour***j*** of ***i*** should contribute **δji** units of work to ***i*** which is the common pool. A negative value for ***δji*** indicates a implies a deficiency, in that case ***j*** will collect – ***δji*** units of work from ***i*** instead of contributing. Similarly, if we look at the work transfer problem from the perspective of a neighbouring processor ***j*** of ***i***, then to achieve perfect load balance between j and its neighbours, processor ***i*** should collect – ***δij*** units of work from j.

In this scheme, when a processor i preempts running out of work, it requests work from its neighbour that has i1 and has the maximum amount of work. A request for work from ***i*** to ***i1*** carries the information **δi1,j**, and the amount of work generated is *min*(**δ*i*1,i – δ*i,i*1**), with the restriction that at least 10% and no more than 50% of the work at i1 is granted. The minimum of the two is taken because we do not want to transfer any extra work that may cause a work transfer in the opposite direction at a later time. If the work request is turned down by processor ***i₁***, say, because ***i₁*** has already granted work to another sink processor in the meantime, then processor ***i*** requests work from the processor with the next most amount of work, and so forth, until it either receives work or has requested from all its neighbour. In the latter case, it waits a certain amount of time, and then resumes requesting work as before. We will refer to work requests meant to effect quantitative load balance as quantitative load requests.

Figure2shows how a sink processor *j3*will request work using the above quantitative load balancing scheme. From the discussion of the NA scheme, we concluded that processor *j3* should actually receive work from processor j4 rather than from **j2** as in the NA scheme, and thus work flows from the heavily loaded neighbourhood of **j4** to the lightly loaded neighbourhood of **j2** via processor **j3**.

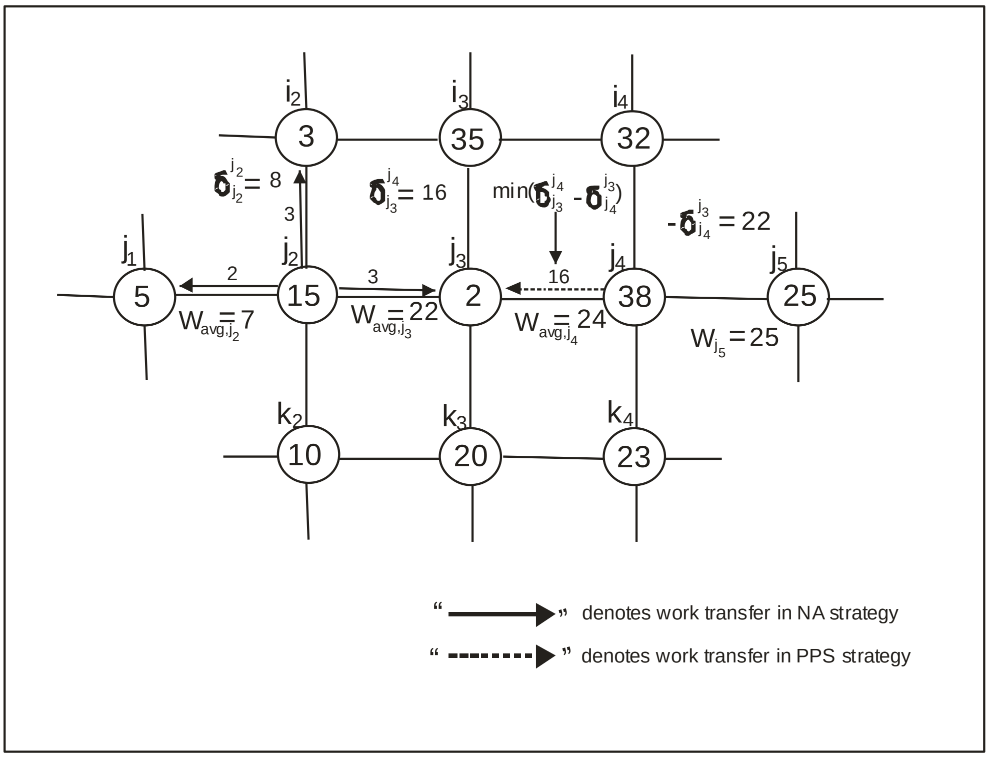
i_2
i_3
i_4
3
35
32
$\delta^{j_2}_{j_2} = 8$
$\delta^{j_4}_{j_3} = 16$
$\min(\delta^{j_4}_{j_3} - \delta^{j_3}_{j_4})$
$-\delta^{j_3}_{j_4} = 22$
3
j_1
j_2
j_3
j_4
j_5
2
3
16
5
15
2
38
25
$W_{avg,j_2} = 7$
$W_{avg,j_3} = 22$
$W_{avg,j_4} = 24$
$W_{j_5} = 25$
k_2
k_3
k_4
10
20
23
" denotes work transfer in NA strategy
" denotes work transfer in PPS strategy

Fig. 2: Work Transfer and Request Process

The NA scheme makes work transfer decisions for processor **j2** considering a processor space comprising its neighbours and itself. i.e. a processor space of radius one. On the other hand, our quantitative load balancing scheme makes work transfer decisions for processor **j3**considering a processor space comprising its neighbours, its neighbour'sneighbour (since the average neighbourhood load of the neighbours is taken into account), and itself, i.e. a processor-space of radius two.

To reduce idling caused by latency between work request and work procurement, processors issue quantitative work requests when starvation is anticipated as follows:We note that at any time the least-cost node in a processor is expanded. Therefore, any decrease in *active-len* below a low threshold implies that the best nodes available are not good enough to generate active nodes and hence this decrease is likely to continue. In this scheme, processors start requesting nodes when active-len is below a certain low threshold. The *acceptor threshold*, and it is decreasing. It is found that this prediction rule works very well in practice. Using such a look-ahead approach, we are able to overlap communication with computation. Moreover, the delay due to transfer of a long message can be mitigated by pipelining the message transfer, i.e., by sending the work in batches.
Basically, the first message unit should be short, so that the processor does not idle long before it receives any work. Subsequent messages (for the same work transfer) can be longer but should not be so long that the preceding message unit gets consumed and the processor idles for an appreciable period of time. For the problem sizes we experimented with, it sufficed to use a short message (one or two nodes) followed by longer messages (each not exceeding 20 nodes).

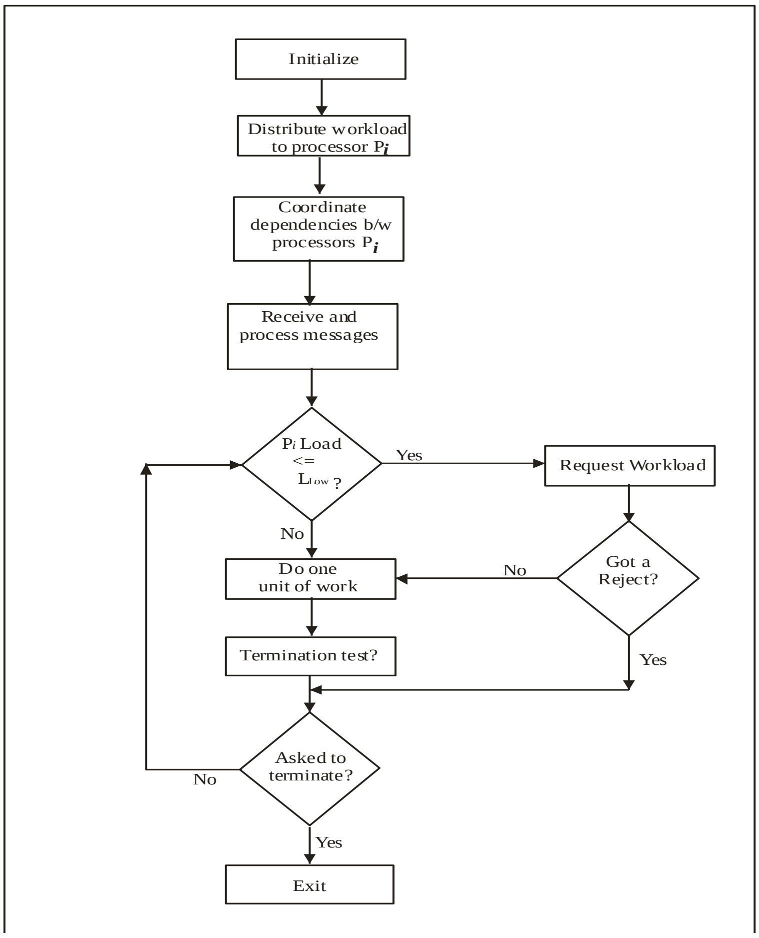
Initialize
Distribute workload to processor P_i
Coordinate dependencies b/w processors P_i
Receive and process messages
P_i Load <= L_{Low} ?
Yes
Request Workload
No
Do one unit of work
No
Got a Reject?
Termination test?
Yes
Asked to terminate?
No
Yes
Exit

Fig. 3: Algorithm Flow for Preemptive Polling Scheme

Note that we perform quantitative load balance only when a processor is about to starve, not all times as in the NA scheme. Thus, our quantitative load balancing scheme overhead is low. Figure 3 shows the process flow for the PPS.

Algorithm 1: Parallel Search with PPS
1. Initialize: *//set all initial conditions for the search*
2. Partition search space: *// Master assigns partitions to available processors*
3. *Coordinate dependencies between processors*
4. *Construct priority queue for partitions local to each process*
5. *Parallel tree search // best-first local search*
Each processor ***i , 0<i<P, executes the following steps:***

```
6.      Report work status: periodically monitor active-len and the threshold cost, and report significant
changes (10% and greater) in them to all neighbours
7.      Receive work report: report and record work status
8. Update j-max and j-best: j-max = neighbor with max active-len value; and j-best = neighbor with least
threshold cost
9. Work request:
        If (no prev work request fromi remains to be serviced) then
        Begin
        If (active-len=0) or <acceptor threshold and is decreasing)
Send a quantitative work request to j-max, along with information $\delta_i^{j-max}$
Else if (lead-node is costlier than threshold cost of j-best)
        Send a qualitative work request to j-best, along with the cost of lead-node
Endif
10. Donate work:if (a quantitative work request is received from neighbour  j) then grant min (δij-δji) (but at
least 10% and not more than 50% of active-len) active nodes in a pipelined fashion
11. Donate work: if (a qualitative work request is received from neighbour j) then grant a few active nodes that
are cheapter than j's lead node
12. Receive work: if (work is received) then insert t nodes received in OPEN
13. Process the node i
14. Apply pruning rules:
15. if (Node I cannot be pruned) then
16.     Create successor of node I on successor function and add to i
17.  Else (Prune node i)
18.   Do termination test?
19. Exit
```

Time Complexity Analysis

Let ***W*** = number of states expanded by serial algorithm
Wp = number of states expanded by parallel algorithm
Search Overhead = ***Wp/W*** describes the overhead due to the order in which states are expanded.
For uninformed search, it is often possible to observe speedup anomalies where:
Wp/W < 1 due to the parallel algorithm searching in multiple regions simultaneously.For informed search the situation is reversed, and the search overhead is added on top of the usual parallel overheads. Since we can compute neither **W** nor **Tp**. We express $\mathbf{T_0}$ in terms of **W** and use $\mathbf{P*Tp = W + T_0}$

Assumptions

Communication subsumes idling (i.e. quantify number of requests).Work can be divided into pieces as long as it is larger than a threshold Є. The work-splitting strategy is reasonable. Whenever work ω is split into two parts $\psi\omega$ and $(1 - \psi)\omega$, there exists an arbitrary small constant $\mathbf{0 < \alpha \leq 0.5}$ such that $\boldsymbol{\psi\omega > \alpha\omega}$ **and** $\boldsymbol{(1 - \psi)\omega > \alpha\omega}$. (In effect, the two pieces are not too imbalanced).
The consequence of the above assumptions: if a processor initially had work ω, then after one split neither processor can have more than $\boldsymbol{(1 - \alpha)\omega}$ work.
Let **V(p)** be the total number of work requests before each process receives at least one work request.
If the largest piece of work at any time is W, then after **V(p)** requests, a processor cannot have more than $\boldsymbol{(1-\alpha)W}$ work (i.e., each processor has been the subject of a split at least once).
After **2V(p)** requests, no more than $\boldsymbol{(1- \alpha)^2}$ W work, and so on.

After **(log1/(1- α) (W/ Є))V(p)** requests, no processor has more work than the threshold Є.Conclusion: total number of work requests is**O(V(p)log W)**

V(P) for Preemptive Polling Scheme (PPS):

Let **F(i,p)** be a state in which ***i*** of the **P** processes have received a request and **P-*i*** have not.
Let **f(i,p)** be the average number of trials required to change from state **F(i , p) to F(P,P)**
V(P) = f(0,p)
F(p,p) = 0
F(i,p) = 1/p (1 + f(i , p)) + p-i/p (1+f(i +1, p)), (p-i /p) f(i,p) = 1 + p-i ,p), f(i,p) = p /p-i + f(i+1,p)
Finally, we get f(o,p) = p = $\sum_{i=1}^{p} \frac{1}{i}$

the harmonic series is roughly **1.69ln p, so**
V(P) = O (plogp)

Isoefficiency Analysis:

For Neighbourhood Averaging (NA)
$T_0 = O(V(p)\log W$
$T_0 = O(V(p)\log W)$
Since $V(P) = O(p^2)$
It follows that:
$W = O(P^2 \log W)$
$= \mathbf{O(P^2 \log(P^2 \log W)}$
$= \mathbf{O(P^2 \log P + P^2 \log\log W)}$
$= \mathbf{O(P^2 \log P)}$

Isoefficiency Analysis
For Global Round-Robin (GRR)
$T_0 = O(V(P)\log W)$
Since $V(P) = O(P)$
It follows that
$W = O(P \log P)$
However, this does not account for the contention at the global counter. The counter is incremented $O(P\log W)$ times in $O(W/P)$ time.
This gives
$W/P = O(P\log W)$
And $W = O(P^2 \log P)$ which is the Isoefficiency

Isoefficiency Analysis
For Preemptive Polling Scheme (PPS)
$T_0 = O(V(P)\log W)$
Since $V(p) = O(P \log P)$
It follows that:
$W = O(P \log P \log W) = O(P \log^2 P)$

From the above analysis, AN has poor performance due to its many requests while GRR suffers from contention.PPS provides is a suitable compromise.

4.1 **Implementation**
Implementation Paradigm
The master-worker paradigm is used here for implementing tree search because a search tree can be easily partitioned into a number of subtrees. In fact, there are not many studies in parallel tree search that use other paradigms other than master-worker. In designing the control strategy of the algorithm, we let the master generate the initial nodes needed during the ramp-up phase, and distribute to the workers who will also have their own local node pools. Although, this paradigm comes with its problems, it is suitable for our implementation because of the size of system used.

Hardware and Test Environment
The physical architecture used is the tests is a network-based system, comprising 16 Sun Ultra 16 Workstations, physically connected by switched 100 Mbit Ethernet. The experiment used a set of systems comprising 2.8GHz Pentium processors and 2GB of RAM. We limited each processor to one parallel process.
To reduce communication demands and alleviate potential network contention, we only use point-to-point local communication functions to implement the parallel system. The parallel search algorithm and test problems were implemented in C++ using MPI protocol for inter-processor communication.

Test Suite
Eight knapsack instances were generated based on the algorithm proposed by]19]. We tested the eight instances of the knapsack in **table 1** by using 4, 8, and 16 processors system. The default algorithm was used to solve the eight instances. The number of nodes generated by the master was 3200 and the number of nodes designated as

a unit of work for 8 processors is 400. Few of the instances were difficult to solve sequentially, but they were all solved to optimality quite easily using just 4 processors. When we increased the number of processors used to 16, the system was able to solve the instances to further reduced time as shown in**table 2**
Table 1 shows basic statistics for the eight instances when solving them sequentially. Column **Item** is the number of items available to be placed in the knapsack. Column **BestSol** is the path cost of the best solution found. Column *Node* is the number of nodes processed and column **NodeLeft** is the number of nodes left when solving serially using the application developed from our algorithm with a time limit of 3000 seconds. We chose 3000 as the time cutoff, since the program can solve most knapsack instances within the range in this amount of time. The last column *Time* shows the wallclock time in seconds used for each instance.

Table 1: Table of the Knapsack Instances with Sequential algorithm

Instance	**Item**	**BestSol**	**Nodes**	**NodeLeft**	**Time**
Input100a	100	19387	19489172	0	2696
Input100b	100	18024	7334000	346716	3000
Input100c	100	18073	6078869	0	1763
Input100d	100	9367	4696000	3290611	3000
Input75a	75	6490	13858959	0	2874
Input75b	75	8271	13433001	0	1233
Input75c	75	8558	18260001	724141	3000
Input75d	75	8210	9852551	0	2672

The table shows that the algorithm could not solve instances **input 100b, input 100d** and **instance 75c** within the 3000 seconds time limit.

We tested the eight instances of the knapsack in table 2 by using 4, 8 and 16 processors system. The default algorithm was used to solve those instances. The number of nodes generated by the master was 3200 and the number of nodes designated as unit of work is 400.

Although a few of these instances were difficult to solve sequentially, they were all solved to optimality quite easily using just 4 processors. As shown in table1, instances **input 100c**, **input 100e**, and **input 75c** are unsolvable with the sequential application in 3000 seconds. However, it took the same program application 128.30 seconds, 67.50 seconds, and 42.43 seconds to solve them respectively when using 4 processors. When we increased the number of processors used to 16, the application was able to solve them in 34.17 seconds, 18.24 seconds, and 11.47 seconds respectively.

Because the results of the 8 instances show similar pattern, we aggregate the results, which are shown in Table 2. The column headers have the following interpretations.

Table 2: Scalability of Solving 8 knapsack instances

P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
4	193057493	0.36%	0.03%	0.02%	52.46	1.00
8	192831731	0.82%	0.09%	0.08%	26.99	0.97
16	192255612	1.67%	0.33%	0.38%	14.08	0.93

- **Nodes**: Is the number of nodes in the search tree. Observing the change in the size of the search tree as the number of processors is increased provides a rough measure of the amount of redundant work being done. Ideally, the total number of nodes explored does not grow as the number of processors is increased and may actually decrease in some cases, due to earlier discovery of feasible solutions that enable more effective pruning.

- **Ramp-up***:* Is the average percentage of total wallclock running time each processor spent idle during ramp-up phase.
- **Idle**: Is the average percentage of total wallclock running time each processor spent idle due to work depletion, i.e., waiting for more work to be provided.
- **Ramp-down**: Is the average percentage of total wallclock running time each processor spent idle during the ramp-down phase.
- **Wallclock:** Is the total wallclock running time (in seconds) for solving the 8 knapsack instances.
- **Eff:** Is the parallel efficiency and is equal to the total wallclock running for solving 10 instances with *p* processors divided by the product of 4 and the total running time with four processors. Note that the efficiency is being measured here with respect to solution time on four processors, rather than one, because of the memory issues encountered in the single-processor runs.

The parallel efficiency is very high if computed in the standard way (see Equation 1). To properly measure the efficiency of our algorithm (application) we use the wallclock on 4 processors as the base for comparison. We therefore define efficiency here as

$$\text{Efficiency} = \frac{\text{Wallclock time using 4 processors x 4}}{\text{Wallclock time using P processor x P}} \text{---------------------------------eqn. 1)}$$

The experiment shows that our algorithm scales well, even when using more processors (i.e. to the number of processors we have used to test it).

Effect of Preemptive Polling Scheme

Again, to test the effectiveness of the dynamic load balancing (preemptive polling scheme) as against a situation where there is no load balancing, we performed different sets of experiments to ascertain this. Table 3shows the results of searching with and without dynamic load balancing enabled. Column balance indicates whether dynamic load balancing scheme was enabled or not. Other columns have the same meaning as those in table 2. As expected, using **PPS** load balancing helped reduce overhead and reduce solution time for all instances. The results reconfirm that dynamic load balancing (in this case, the Preemptive Polling Scheme) is important to the performance of a parallel search algorithm.

Table 3: The Effect of PPS load balancing on knapsack instances

P	Balance	Node	Ramp-up	Idle	Ramp-down	Wallclock
4	No	10771651	0.43%	0.00%	49.61%	87.60
	Yes	16503223	0.57%	0.07%	0.16%	26.02
8	No	13791025	0.44%	0.00%	39.80%	70.18
	Yes	11535858	1.00%	1.00%	0.08%	18.01

III. Conclusion

Parallel programming field lacks a common development direction. Tree search problems are examples of the formulation of Discrete Optimization Problems (DOP). We have proposed a novel dynamic load balancing scheme for parallel search called Preemptive Polling Scheme(PPS). Comparative analysis of PPS with other studied dynamic load balancing schemes suggests that PPS is a desirable balance. An appropriate method for the sharing of workload is a critical choice to make in parallel search problems. The expressive power of the PPS has been investigated in-depth both theoretically and empirically.The experiment shows that the introduction of PPS improves the overall load balancing of the system, and does not introduce significant overhead, in terms of execution time, even if it requires the exchange of a higher number of messages between nodes. Future expansions to this include extensive/expansion of the number of nodes.

References

[1]. Pfister, G. (1998). In Search of Clusters, Prentice Hall PTR, NJ, 2nd Edition, NJ 1998

[2]. Aleem, A., Terek, H. and Irfan, A. (2008). A Framework for Fair and Reliable Resource Sharing in Distributed Systems. Springer Publisher, ISBN 978-0-387-79447-1

[3]. Kunz, T. (1991). The influence of different workload descriptions on a heuristic load balancing scheme, IEEE Trans on Software Engineering, 17 (7) pp 725-730.

[4]. Xu, Y. (2003). Optimal Parameters for Load balancing using the Diffusion Method in k-ary n-cube Networks, Information Processing Letters pp181-187.

[5]. Monien, B. and Luling, R. (1992). Load Balancing for Distributed Branch-and-Bound Algorithms. Sixth Int'l Par. Proc. Symp. pp 543-548.

[6]. Padberg, M. and Rinaldi, G. (1991). A branch-and-cut algorithm for the resolution of large-scale traveling salesman problems. SIAM Review, 33:60.100.

[7]. Sanders, P. (1998). Tree shaped computations as a model for parallel applications. Technical Report iratr

[8]. Korf, R. E. (1999); Artificial intelligence search algorithms. In Algorithms and Theory of Computation Handbook. CRC Press.

[9]. Cormen, T., Leiserson, C., and Rivest, R. (2001). Introduction to Algorithms, volume 2. The MIT Press, Massachusetts, USA.

[10]. Russell, S. and Norvig, P. (2003). Artificial Intelligence: A Modern Approach. Pearson Education, Inc., USA, 2nd edition.

[11]. Chapman, B., Jost, G. and R. van der Pas (2007). Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, USA, 1st edition.

[12]. Gropp, W., E. Lusk, E. and Skjellum, A. (2003). Using MPI. MIT Press, Cambridge, MA, USA, 3rd edition.

[13]. Dan, B. (2006). AMMPI: Active Message Passing MPI. MPI Protocol Conf.

[14]. Geist, A., Beguelin, A., and Jiang, W. (2006). PVM-Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, USA.

[15]. N.R. Mahapatra and S. Dutt, "Scalable Duplicate Pruning Strategies for Parallel A* Graph Search, "Fifth IEEE Symp. On Par. And Distr. Proc'g pp 290-297, Dec. 1993.

[16]. V. A. Saletore, "A Distributed and Adaptive Dynamic Load Balancing Scheme for Parallel Processing of Medium-Grain Tasks," Proc. Fifth Distributed Memory Conference, 1990

[17]. A. B. Sinha and L. V. Kale, "A Load Balancing Strategy for Prioritized Execution of Tasks," Seventh Int'l Par. Proc. Symp., pp230-237

[18]. Grama, A., Gupta, A., and Kumar, V. (1993). Isoefficiency: Measuring the scalability of parallel algorithms and architectures. IEEE Parallel Distrib. Technol., 1(3):12. 21.

[19]. Martello, S. and Toth, P. (1977). Knapsack Problems: Algorithms and Computer Implementation. John Wiley & Sons, Inc., USA, 1st edition.