Parallelization on Multi-Core Machines using Global Address Languages

Thesis
Submitted in partial fulfillment of the requirements of
BITS C421T Thesis

By

Parvesh Jain
ID No.: 2007A7TS128P

Under the supervision of

Prof. R.K. Shyamasundar
Senior Professor
School of Technology & Computer Science
Tata Institute of Fundamental Research – Mumbai (India)

Tata Institute of Fundamental Research, Mumbai

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
(RAJASTHAN, INDIA)

May 9, 2011

Acknowledgments

I would like to express sincere gratitude to Dr. A. K. Das, Dean, Research and Consultancy Division (RCD) for providing the opportunity of doing a thesis at Tata Institute of Fundamental Research (TIFR) – Mumbai (India). I express my gratitude to project guides Prof. R.K. Shyamasundar , Senior Professor, School of Technology & Computer Science, TIFR - Mumbai and Mr. Ankur Narang, Research Lead HPA(High Performance Analytics, IBM India Research Laboratory, New Delhi ,India for extending their professional support, continuous guidance and encouragement throughout the project. I would also like to thank Mr. John Barreto, Secretary, School of Technology & Computer Science for assisting me in all the administrative work related to thesis. Without their help and motivation the thesis would have been a distant reality.

CERTIFICATE

This is to certify that the Thesis entitled, **Parallelization on Multi-Core Machines using Global Address Languages** and submitted by **Parvesh Jain** bearing ID No. 2007A7TS128P in partial fulfilment of the requirement of BITS C421T/422T Thesis embodies the work done by him under my supervision.

RK Shyamasundar

Date: 9th May, 2011

Signature of the Supervisor

Name: R. K. Shyamasundar

Designation: Senior Professor

School of Technology & Computer Science

Tata Institute of Fundamental Research, Mumbai

Abstract

The partitioned global address space (PGAS) is a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each processor. The novelty of PGAS is that the portions of the shared memory space may have an affinity for a particular thread, thereby exploiting locality of reference. The PGAS model is the basis of Intel Cilk Plus ,Cilk , Co-array Fortran, Titanium, Fortress, Chapel and X10.In initial part of my thesis I worked on a number of PGAS and explored them. In subsequent part I tried to observe an Efficient Shared Memory and Distributed Memory parallelization of Graph Algorithms like Depth First Search & Breadth First Search on a SMP architecture using IBM X10.

Later I also worked on Distibuted workstealing on multiprocessors trying to develop a Deadlock free Affinity Driven Distributed Scheduling Algorithm for Parallel Computations**.**

Table of Contents

Introduction **5**

Chapter 1 …... **7**

Partitioned Global Address Space
languages…………….. 7

Chapter 2………….. **10**

Efficient Shared Memory and Distributed Memory
Parallelization of Depth First Search & Breadth
First Search on SMP architecture using IBM X10…………………………………………….10

Parallel Breadth First Search……………………………………………………………..10
Parallel Depth First Search……………………………………………………………….17

Chapter 3………….. **21**

Deadlock free Affinity Driven Distributed
Scheduling Algorithm for Parallel Computations ……………………………………………21

Conclusion and Future
Work…….. **27**

Bibliography &
References... **29**

Introduction

The exascale computing roadmap has highlighted efficient locality oriented scheduling in runtime systems as one of the most important challenges (”Concurrency and Locality” Challenge). Massively parallel many core architectures have NUMA characteristics in memory behavior, with a large gap between the local and the remote memory latency. Unless efficiently exploited, this is detrimental to scalable performance.

Languages such as Cilk, Cilk Plus (Intel), X10(IBM) and Chapel (Cray) are based on partitioned global address space paradigm. They have been designed and implemented for higher productivity and performance on many-core massively parallel platforms. These languages have in-built support for initial placement of threads (also referred as activities) and data structures in the parallel program.

PGAS programming models offer HPC programmers an abstracted shared address space, which simplifies programming, while exposing data/thread locality to enhance performance. This can facilitate the development of productive programming languages that can reduce the time to solution, i.e. both development time and execution time.

Locality comes implicitly with the program. The run-time systems of PGAS languages need to provide efficient algorithmic scheduling of parallel computations with medium to fine grained parallelism. For handling large parallel computations, the scheduling algorithm (in the run-time system) should be designed to work in a distributed fashion. This is also imperative to get scalable performance on many core architectures. Further, the execution of the parallel computation happens in the form of a dynamically unfolding execution graph.
It is difficult for the compiler to always correctly predict the structure of this graph and hence perform correct scheduling and optimizations, especially for data-dependent computations. Therefore, in order to schedule generic parallel computations and also to exploit runtime execution and data access patterns, the scheduling should happen in an online fashion. Moreover, in order to mitigate the communication overheads in scheduling and the parallel computation, it is essential to follow affinity inherent in the computation. Simultaneous consideration of these factors along with low time and message complexity, makes distributed scheduling a very challenging problem.

Algorithms to search a graph in a breadth-first manner have been studied for over 50 years.

The first breadth-first search (BFS) algorithm was discovered by Moore while studying the problem of finding paths through mazes. Lee independently discovered the same algorithm in the context of routing wires on circuit boards. A variety of parallel BFS algorithms have since been explored .Some of these parallel algorithms are work efficient, meaning that the total number of operations performed is the same to within a constant factor as that of a comparable serial algorithm. That constant factor, which we call the work efficiency, can be important in practice, but few if any papers actually measure work efficiency.

Depth-first search is a general technique used in Artificial Intelligence for solving a variety of problems in planning, decision making, theorem proving, expert systems etc. It is also used under the name of backtracking to solve various combinatorial problems and constraint satisfaction problems. Execution of prolog program can be viewed as depth-first search of a proof tree. Iterative deepening DFS algorithms are used to solve discrete optimization problems and for theorem proving. A major advantage of depth-first strategy is that it requires very little memory. Since many of the problems solved by DFS are highly computation intensive, there has been great interest in developing parallel versions of depth-first search.

Chapter 1

Partitioned Global Address Space Languages

PGAS programming models offer HPC programmers an abstracted shared address space, which simplifies programming, while exposing data/thread locality to enhance performance. This can facilitate the development of productive programming languages that can reduce the time to solution, i.e. both development time and execution time. I started with my study with **Cilk** (MIT), then moved to **Cilk Plus** (Intel) and finished with **Chapel** (Cray) and **X10**(IBM).Following are the features I was able to explore and experiment with.

- **Cilk**: It is a linguistic and runtime technology for algorithmic multithreaded programming developed at MIT. The philosophy behind Cilk is that a programmer should concentrate on structuring her or his program to expose parallelism and exploit locality, leaving Cilk's runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. The Cilk runtime system takes care of details like load balancing, synchronization, and communication protocols. Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance. Important milestones in Cilk technology include the original Cilk-1, which provided a provably efficient work-stealing runtime support but little linguistic support; the later Cilk-5, which provided simple linguistic extensions for multithreading to ANSI C; and the commercial Cilk++, which extended the Cilk model to C++ and introduced "reducer hyper objects" as an efficient means for resolving races .Cilk is an especially effective platform for programming "irregular" applications such as sparse numerical algorithms, N-body simulations, graph-theory applications, backtracking search, and cache-efficient stencil computations.

- **Intel Cilk Plus**: Cilk Plus is a general-purpose programming language designed for multithreaded parallel computing. On July 31, 2009, Cilk Arts, producers of Cilk++ programming language, announced that its products and engineering team were now part of Intel Corp. Intel and Cilk Arts integrated and advanced the technology further resulting in a September 2010 release of **Intel Cilk Plus**. Intel Cilk Plus adopts simplifications, proposed by Cilk Arts in Cilk++, to eliminate the need for several of the original Cilk keywords while adding the ability to spawn functions and to deal with variables involved in reduction operations. Intel Cilk Plus differs from Cilk and Cilk++ by adding array extensions, being incorporated in a commercial compiler (from Intel), and compatibility with existing debuggers.

- **Chapel**: It is a new parallel programming language being developed by Cray Inc. as part of the DARPA-led High Productivity Computing Systems program (HPCS). Chapel is designed to improve the productivity of high-end computer users while also serving as a portable parallel programming model that can be used on commodity clusters or desktop multicore systems. Chapel strives to vastly improve the programmability of large-scale parallel computers while matching or beating the performance and portability of current programming models like MPI.

  Chapel supports a multithreaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. Chapel's locale type enables users to specify and reason about the placement of data and tasks on target architecture in order to tune for locality. Chapel supports global-view data aggregates with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. In contrast to many previous higher-level parallel languages, Chapel is designed around a multi resolution philosophy, permitting users to initially write very abstract code and then incrementally add more detail until they are as close to the machine as their needs require. Chapel supports code reuse and rapid prototyping via object-oriented design, type inference, and features for generic programming.

  Chapel was designed from first principles rather than by extending an existing language. It is an imperative block-structured language, designed to be easy to learn for users of C, C++, FORTRAN, Java, Perl, Matlab, and other popular languages.

- **X10**: IBM Research is developing the open-source X10 programming language to provide a programming model that can address the architectural challenge of multiples cores, hardware accelerators, clusters, and supercomputers in a manner that provides scalable performance in a productive manner. The project leverages over six years of language research funded, in part, by the DARPA/HPCS program.

The X10 programming language is organized around four basic principles of asynchrony, locality, atomicity, and order that are developed on a type-safe, class-based, object-oriented foundation. This foundation is robust enough to support fine-grained concurrency, Cilk-style fork-join programming, GPU programming, SPMD computations, phased computations, active messaging, MPI-style communicators, and cluster programming.

X10 is a type-safe, parallel object-oriented language. It targets parallel systems with multi-core SMP nodes interconnected in scalable cluster configurations. A member of the Partitioned Global Address Space (PGAS) family of languages, X10 allows the programmer to explicitly manage locality via Places. Lightweight activities embodied in async, constructs for termination detection (finish) and phased computation (clocks), and the manipulation of global arrays and data structures.

Chapter 2

Efficient Shared Memory and Distributed Memory parallelization of Depth First Search & Breadth First Search on SMP architecture using IBM X10

Parallel Breadth First Search

I worked on developing a multithreaded implementation of breadth-first search (BFS) of a sparse graph using the IBM X10 for single place SMP architecture. Parallel Breadth First Search(PBFS) aims at achieving high work-efficiency by using a novel implementation of a multi set data structure, called a "bag," in place of the FIFO queue usually employed in serial breadth-first search algorithms.

Given a graph G = (V, E) with vertex set V = V (G) and edge set E = E (G), the BFS problem is to compute for each vertex v belonging to V the distance v. dist that v lies from a distinguished source vertex v0 belonging to V. We measure distance as the minimum number of edges on a path from v0 to v in G. For simplicity in the statement of results, we shall assume that G is connected and directed, although the algorithms we shall explore apply equally as well to unconnected graphs, digraphs, and multigraphs.

Classical serial algorithm uses a FIFO queue as an auxiliary data structure. The FIFO can be implemented as a simple array with two pointers to the head and tail of the items in the queue. Enqueueing an item consists of incrementing the tail pointer and storing the item into the array at the pointer location. Dequeueing consists of removing the item referenced by the head pointer and incrementing the head pointer. Since these two operations take only Θ (1) time, the running time of SERIAL-BFS is Θ (V +E). Moreover, the constants hidden by the asymptotic notation are small due to the extreme simplicity of the FIFO operations. Although efficient, the FIFO queue Q is a major hindrance to parallelization of BFS. Parallelizing BFS while leaving the FIFO queue intact yields minimal parallelism for sparse graphs — those for which $|E| \approx |V|$. The reason is that if each ENQUEUE operation must be serialized, the span1 of the computation — the longest serial chain of executed instructions in the computation — must have length $\Omega(V)$. Thus, a work-efficient algorithm — one that uses no more work than a comparable serial algorithm — can have parallelism — the ratio of work to span — at most O ((V + E)/V) = O (1) if |E | = O (V) [2]

Replacing the FIFO queue with another data structure in order to parallelize BFS may compromise work efficiency, however, because FIFO s are so simple and fast We have devised a multiset data structure called a bag, however, which supports insertion essentially as fast as a FIFO, even when constant factors are considered. In addition, bags can be split and unioned efficiently.

THE PBFS ALGORITHM

PBFS uses layer synchronization to parallelize breadth-first search of an input graph G. Let v_0 belonging to V (G) be the source vertex, and define layer **d** to be the set V_d belonging to V (G) of vertices at distanced from v_0. Thus, we have $V_0 = \{v_0\}$. Each iteration processes layer d by checking all the neighbors in a parallel manner of vertices in V_d for those that should be added to V_{d+1}.

PBFS implements layers using an unordered-set data structure, called a bag, which provides the following operations:

- bag= BAG -CREATE (): Create a new empty bag.

- BAG -INSERT (bag, x): Insert element x into bag.

- BAG -UNION (bagl, bag2): Move all the elements from bag2 to bagl and destroy bag2.

- bag2 =BAG -SPLIT (bagl): Remove half (to within some constant amount GRAINSIZE of granularity) of the elements from bagl and put them into a new bag bag.

Pseudo Code for the PBFS Algorithm

```
PBFS (G, v0)
1.  Finish for each vertex v belonging to V (G) – {v_0} // Parallely initialize the distance for each
    node.
2.  async {
3.  v.dist = INFINITE
4.  }}
5.  v_0.dist = 0
6.  d = 0
```

```
7.  B_0 = BAG-CREATE()
8.  BAG-INSERT(B_0,v_0)
9.  while not BAG-IS-EMPTY(B_d)
10. {
11. B_{d+1} = new BAG-CREATE()
12. PROCESS-LAYER(B_d ,B_{d+1},d)
13. B_d  = B_{d+1}
14. }
```

PROCESS-LAYER (in-bag, out-bag,d)

```
1.  if BAG-SIZE(in-bag)  < GRAINSIZE
2.  for each u belonging to in-bag
3.  Finish for each v belonging to Adj[u] //Parallely explore the neighbors of a node .
4.  async {
5.  ob = new BAG-CREATE()
6.  if v.dist == INFINITE
7.  v.dist = d +1
8.  BAG-INSERT(ob,v)
9.  BAG-UNION(out-bag,ob)
10. return
11. }}
12. new-bag = BAG-SPLIT(in-bag)
13. finish{
14. ob = new BAG-CREATE()
15. async{
16. PROCESS-LAYER(new-bag,ob,d)
17. }
18. PROCESS-LAYER(in-bag,out-bag,d)
19. BAG-UNION(out-bag,ob)
20. }
```

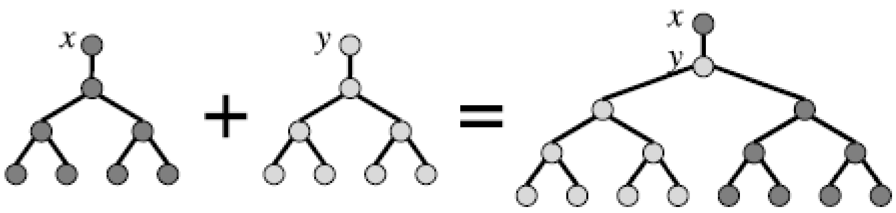
THE BAG DATA STRUCTURE

This section describes the bag data structure for implementing a dynamic unordered set. We first describe an auxiliary data structure called a "pennant." We then show how bags can be implemented using pennants, and we provide algorithms for BAGCREATE, BAG-INSERT, BAG-UNION, and BAG-SPLIT. Finally, we discuss some optimizations of this structure that PBFS employs.

Pennants

A **pennant** is a tree of 2^k nodes, where k is a nonnegative integer. Each node x in this tree contains two pointers x. left and x. right to its children. The root of the tree has only a left child, which is a complete binary tree of the remaining elements. Two pennants x and y of size 2k can be combined to form a pennant of size 2^{k+1} in O(1) time using the following PENNANT-UNION function

```
PENNANT-UNION(x,y)
1 y. right = x. left
2 x. left = y
3 return x
```



Two pennants each of size 2^k can be unioned in constant time to form a pennant of size 2^{k+1}

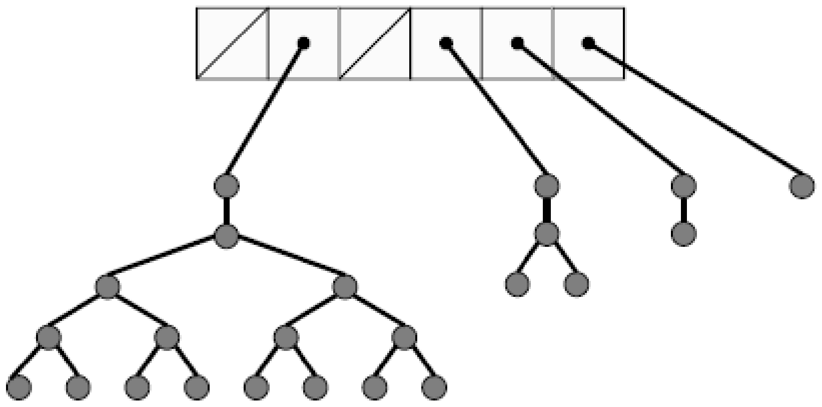
The function PENNANT-SPLIT performs the inverse operation of PENNANT-UNION in O (1) time. We assume that the input pennant contains at least 2 elements.

```
PENNANT-SPLIT(x)
1 y = x. left
2 x. left = y. right
3 y. right = NULL
4 return y
```

Each of the pennants x and y now contains half the elements.

Bags

A bag is a collection of pennants, no two of which have the same size. PBFS represents a bag S using a fixed-size array S [0 . . . r], called the backbone, where 2^{r+1} exceed the maximum number of elements ever stored in a bag. Each entry S[k] in the backbone contains either a null pointer or a pointer to a pennant of size 2^k. The function BAG-CREATE allocates space for a fixed-size backbone of null pointers, which takes O(r) time. This bound can be improved to O (1) by keeping track of the largest nonempty index in the backbone



A bag with 23 = 0101112 elements

The BAG-INSERT function employs an algorithm similar to that of incrementing a binary counter. To implement BAG-INSERT, we first package the given element as a pennant x of size 1. We then insert x into bag S using the following method.

```
BAG-INSERT(S, x)
1 k = 0
2 while S[k] _= NULL
3 x = PENNANT-UNION(S[k], x)
4 S [k++] = NULL
5 S [k] = x
```

The analysis of BAG-INSERT mirrors the analysis for incrementing a binary counter. Since every PENNANT-UNION operation takes constant time, BAG-INSERT takes O (1) amortized
time and O (lgn) worst-case time to insert into a bag of n elements. The BAG-UNION function uses an algorithm similar to ripple carry addition of two binary counters. To implement BAG-UNION, we first examine the process of unioning three pennants into two pennants, which operates like a full adder. Given three pennants x, y, and z, where each either has size 2^k or is empty, we can merge them

to produce a pair of pennants (s, c), where s has size 2^k or is empty, and c has size 2^{k+1} or is empty. The following table details the function FA(x, y, z) in which (s, c) is computed from (x, y, z), where 0 means that the designated pennant is empty, and 1 means that it has size 2^k:

x	y	z	s	c
0	0	0	NULL	NULL
1	0	0	x	NULL
0	1	0	y	NULL
0	0	1	z	NULL
1	1	0	NULL	PENNANT-UNION(x,y)
1	0	1	NULL	PENNANT-UNION(x,z)
0	1	1	NULL	PENNANT-UNION(y,z)
1	1	1	x	PENNANT-UNION(y,z)

With this full-adder function in hand, BAG-UNION can be implemented as follows:

```
BAG-UNION (S1, S2)
1 y = NULL // the “carry” bit.
2 for k = 0 to r
3 (S1 [k], y) = FA (S1 [k], S2 [k], y)
```

Because every PENNANT-UNION operation takes constant time, computing the value of FA(x, y, z) also takes constant time. To compute all entries in the backbone of the resulting bag takes Θ(r) time. This algorithm can be improved to Θ (lgn), where n is the number of elements in the smaller of the two bags, by maintaining the largest nonempty index of the backbone of each bag and unioning the bag with the smaller such index into the one with the larger.

The BAG-SPLIT function operates like an arithmetic right shift:

```
BAG-SPLIT (S1)
1 S2 = BAG-CREATE ()
2 y = S1 [0]
3 S1 [0] = NULL
4 for k = 1 to r
5 if S1 [k] _= NULL
6 S2 [k–1] = PENNANT-SPLIT (S1 [k])
7 S1 [k–1] = S1 [k]
8 S1 [k] = NULL
9 if y! = NULL
```

```
10 BAG-INSERT (S1, y)
11 return S2
```

Because PENNANT-SPLIT takes constant time, each loop iteration in BAG-SPLIT takes constant time. Consequently, the asymptotic runtime of BAG-SPLIT is O(r). This algorithm can be improved to Θ (lgn), where n is the number of elements in the input bag, by maintaining the largest nonempty index of the backbone of each bag and iterating only up to this index.

Parallel Depth First Search

I worked on a parallel formulation of depth-first search of state space trees using the IBM X10 for a single place SMP architecture which retains the storage efficiency of sequential depth-first search. At the heart of our parallel formulation is a dynamic work distribution scheme that divides the work between different activities (light weight threads).The effectiveness of the parallel formulation is strongly influenced by the dynamic work distribution scheme.

The main advantage of depth-first search over the other search techniques is its low storage requirements. For most other search techniques (such as breadth first and best first) the storage requirement is exponential in the length of the solution path, whereas for depth first search, the storage requirement is linear in the depth of the space searched. But sequential depth-first search has the following major drawback.

- ❖ If the search space to the left of the first goal node is infinite (or very large) then search would never terminate (or take very long time).

One possible way of handling this problem is to have a bound on the depth of the space searched sequentially .As the current depth exceeds this bound we can start parallel depth first search on some unexplored nodes.

Parallel Depth First Search Algorithm

We parallelize DFS by sharing the work to be done among activities. Each activity searches a disjoint part of the search-space in a depth-first fashion.Whenver work to be done by an activity exceeds the threshold, its work is divided and the divided part of the work is given to a new activity for parallel depth first search. When a goal node is found, all of them quit. If the search space is finite and has no solutions, then eventually all the activities would run out of work and the (parallel) search will terminate.

Since each activity searches the space in a depth-first manner, the (part of) state space to be searched is efficiently represented by a stack. The depth of the stack is the depth of the node being currently explored; and each level of the stack keeps track of the untried alternatives. Each activity maintains its own local Stack on which it executes DFS.When the local stack work exceeds the given threshold, a part of the work is given to a new activity. In our implementation, at the start of the algorithm single activity explores the root node and keeps on storing the unexplored nodes in a local stack, From then on, the search space is divided and distributed among various activities.

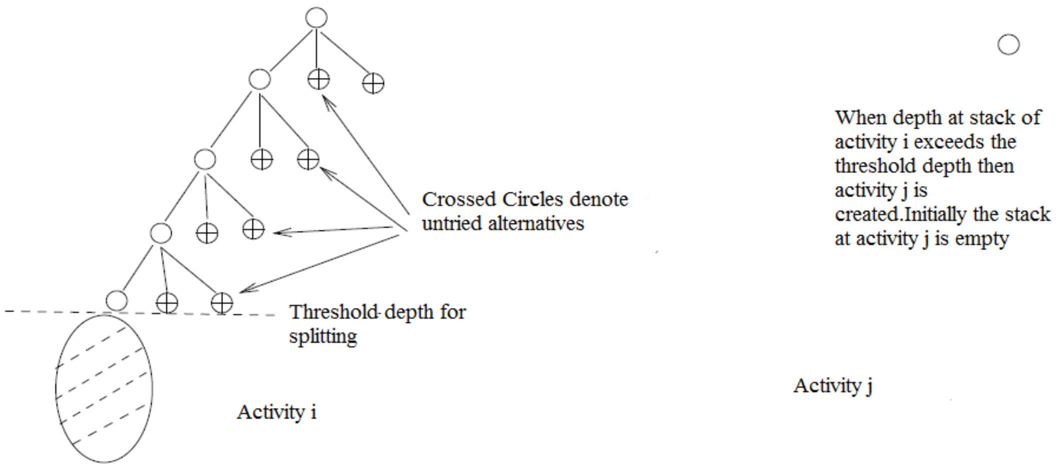
The basic driver routine in each of the activities is given below. A denotes the activity, and $stack_A$ denotes the local stack of the activity.

```
Parallel DFS :( Activity A, stack $stack_A$)
while(not terminated) do
{
    while($stack_A$ not empty)
    {
        n = $stack_A$.pop()
        terminated = n.checkforSolution()
        if (terminated)
        {
            return
        }
        for all nodes $n_0$ adjacent to n
        {
            $stack_A$.push($n_0$)
        }
        if($stack_A$.size > THRESHOLD)
        {
            async
            {
                $stack_{Anew}$ = new stack()
                $stack_{Anew}$ = stack.splitStack(splitsize)  //Divide the work
                Parallel DFS:(Activity Anew,$stack_{Anew}$)
            }
        }
    }
}
```

Procedure split in stack class is used for splitting the work in the current stack to a new stack corresponding to the value of splitsize.

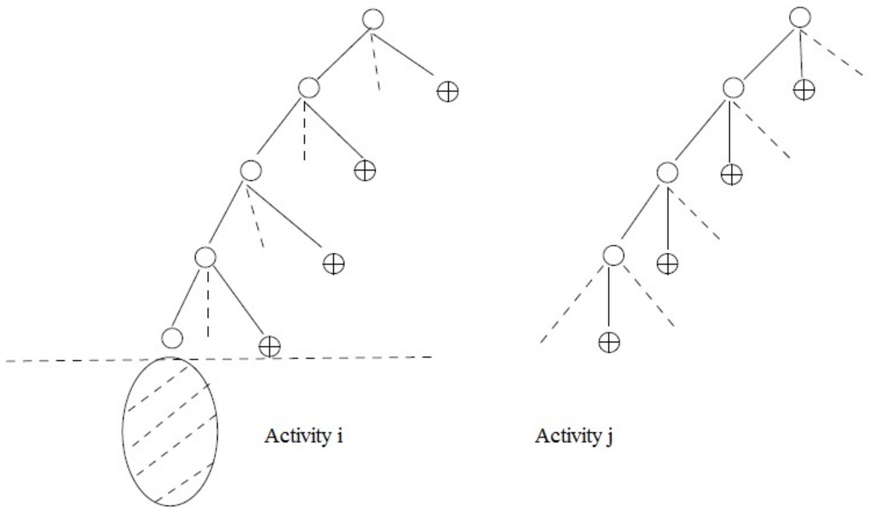
splitStack: ($stack_{old}$, splitSize)

```
1. $stack_{new}$ = new stack()
2. while ($stack_{new}$.size < splitSize)
3. {
4.     $stack_{new}$.push($stack_{old}$.removeNode()) // Node can be removed from a   //stack in no. of ways
       in terms of position of the node removed
5. }
6. return $stack_{new}$
```



Crossed Circles denote untried alternatives
Threshold depth for splitting
Activity i
When depth at stack of activity i exceeds the threshold depth then activity j is created.Initially the stack at activity j is empty
Activity j

Stack of Activity i before splitting



Stacks of Activity i and j after splitting

Important parameters of the Parallel Formulation

Two parameters of the algorithm are important for performance on a given architecture and a given problem

1. The splitting strategy: When a work transfer is made, work in the donor's tack is split into two stacks one of which is given to the requester. In other words, some of the nodes (i.e., alternatives) from the donor's stack are removed and added to the requester's stack. Intuitively it is ideal to split the stack into two equal pieces (called ½ split).If the work given out is too small, then the requester would become idle too soon. If the work given out is too large, then the donor will become idle too soon.
   Many strategies for removing nodes is possible
   a. Pick up some nodes near the root.
   b. Pick up some nodes near the cut off depth.
   c. Pick up half of all the available nodes above the threshold depth.

The suitability of a splitting strategy is dependent upon the nature of the search space. If the search tree is uniform, then both strategies 1 and 3 would result in a good splitting. If the search space is highly irregular, then only Strategy 2 results in a good splitting.

2. Threshold depth: On a shared-memory system, the threshold depth serves the additional purpose of transferring work without interrupting the target processor, as the target processor can continue working on nodes below the threshold depth should be chosen so that the work done below threshold depth takes much longer time when compared to the stack transfer time, so that a work transfer occurs without significantly slowing down the target activity

Chapter 3

Deadlock free Affinity Driven Distributed Scheduling Algorithm for Parallel Computations

For handling large parallel computations, the scheduling algorithm (in the run-time system) should be designed to work in a distributed fashion. This is also imperative to get scalable performance on many core architectures. Further, the execution of the parallel computation happens in the form of a dynamically unfolding execution graph. It is difficult for the compiler to always correctly predict the structure of this graph and hence perform correct scheduling and optimizations, especially for data-dependent computations. Therefore, in order to schedule generic parallel computations and also to exploit runtime execution and data access patterns, the scheduling should happen in an online fashion. Moreover, in order to mitigate the communication overheads in scheduling and the parallel computation, it is essential to follow affinity inherent in the computation. Simultaneous consideration of these factors along with low time and message complexity, makes distributed scheduling a very challenging problem.
Specifically, we address the following affinity driven distributed scheduling problem:

Given:

- An input computation DAG that represents a parallel multi-threaded computation with fine to medium grained parallelism. The DAG is defined as follows:
    - Each node in the DAG is a basic operation such as and/or/add etc. and is annotated with a place identifier which denotes where that node should be executed.
    - Each edge in the DAG represents one of the following: (i) spawn of a new thread or, (ii) sequential flow of execution or, (iii) synchronization dependency between two nodes.
    - The DAG is a terminally strict parallel computation DAG (synchronization dependency edge represents an activity waiting for the completion of a descendant activity).
- A cluster of n SMPs as the target architecture on which to schedule the computation DAG. Each SMP also referred as place has fixed number (m) of processors and memory. The space per place is bounded as defined by the target architecture. The cluster of SMPs is referred as the multi-place setup.

Determine: An online schedule for the nodes of the computation DAG in a distributed fashion that ensures the following:

- Exact mapping of nodes onto places as specified in the input DAG.
- Physical deadlock free execution.
- Low time and message complexity for execution.
- Execute within the bounded space per place

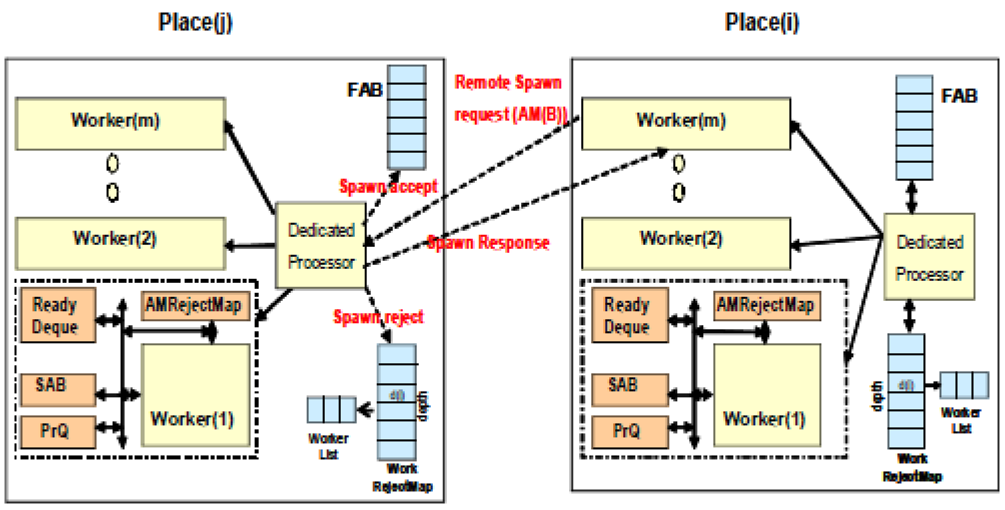
In this project, we worked on the design of a novel affinity driven, online, distributed scheduling algorithm with low time and message complexity while guaranteeing deadlock free execution. The algorithm assumes initial placement annotations on the given parallel computation with consideration of load balance across the places. The algorithm controls the online expansion of the computation DAG based on available space. The scheduling algorithm carefully manages space for execution in a distributed fashion using computation depth upper-bound based ordering of threads. This distributed deadlock avoidance strategy ensures deadlock free execution of the parallel computation and we prove this formally in this paper. Further, our algorithm employs an efficient remote spawn and reject handling mechanism across places for ensuring affinity. Randomized work stealing within a place helps in load balancing.

Distributed Scheduling Algorithm

Consider a terminally strict place-annotated computation DAG. Our distributed scheduling algorithm schedules activities with affinity, at only their respective places. Within a place, work stealing is enabled to allow load-balanced execution of the computation sub-graph associated with that the place. The computation DAG unfolds in an online fashion in a breadth-first manner across places when the affinity driven activities are pushed onto their respective remote places. Within a place, the online unfolding of the computation DAG happens in a depth-first manner to enable efficient space and time execution.

Each place maintains a Fresh Activity Buffer (FAB) which is managed by a dedicated processor (which is a regular core/processor other than workers for a distributed environment) at that place. An activity that has affinity for a remote place is pushed (referred as work-pushing) into the FAB at that place. If the remote place FAB has sufficient space then it accepts the remote activity and it gets inserted in the remote FAB. Else, it rejects the remote spawn and the information about the rejected activity and worker is stored in both the source worker and the destination place. The worker which initiated the remote spawn re-attempts this spawn only when sufficient space is guaranteed in the remote FAB. This helps in limiting the number of messages for a successful remote spawn.

For load balancing within a place, an idle worker at a place will attempt to randomly steal work from other workers at the same place (randomized work stealing). When a worker attempts to steal an activity from the victim worker, then it first checks whether it has sufficient space to execute that activity using the space it has available. The space required to execute an activity at depth d_i, is dependent on the remaining depth (D_{max} - d_i) with respect to the maximum depth (D_{max}) of the computation tree. The steal attempt is successful and the activity is stolen, if the thief worker has this space demanded by the stolen activity, else the thief makes another steal attempt. Note, that an activity which is pushed onto a place can move between workers at that place (due to work stealing) but cannot move to another place and thus obeys affinity at all times.



Distributed Data Structures & Spawn/Reject Mechanism Across Places

Due to limited space on real systems, this distributed scheduling algorithm has to limit online breadth first expansion of the computation DAG while minimizing the impact on execution time and simultaneously providing deadlock freedom guarantee. To achieve this, our algorithm uses a novel distributed deadlock avoidance scheme. Due to space constraints at each place in the system, the activities can be stalled due to lack of space. The algorithm keeps track of stack space available on the system and that required by activities for execution (heap space is not considered for simplicity of discussion). The space required by an activity, u, is bounded by the maximum stack space needed for its execution, i.e. ((D_{max} - D_u) .S_{max}), where, D_{max} is the maximum activity depth in the computation tree, D_u is the depth of u in the computation tree, S_{max} is the size of the largest activation frame in the

computation. The algorithm follows depth based ordering of computations for execution by allowing the activities with higher depth
on a path to execute to completion before the activities with lower depth on the same path. This happens in a distributed fashion. Both during work-pushing and intra-place work stealing, each place and worker checks for availability of stack space for execution of the activity, respectively. Due to depth based ordering, only bounded numbers of paths in the computation tree are expanded at any point of time. This bound is based on the available space in the system.
Using this distributed deadlock avoidance scheme, the system always has space to guarantee the execution of a certain number of paths that can vary during the execution of the computation DAG.

An activity can be in any of the multiple stalled states (local-stalled or remote-stalled or depend-stalled) during the execution of the scheduling algorithm. When an activity is stalled due to lack of space at a worker, it moves into local-stalled state. When an activity is stalled as it cannot be spawned onto a remote place, it moves into remote-stalled state. An activity that is stalled due to synchronization dependencies, it moves into depend-stalled state. We assume that the maximum depth of the computation tree (in terms of number of activities), D_{max}, can be estimated fairly accurately prior to the execution, from the parameters used in the input parallel computation. D_{max} value is used in our distributed scheduling algorithm to ensure physical deadlock free execution.

Distributed Data-Structures & Algorithm Design

The distributed data structures used in the scheduling algorithm are described below. Each worker at a place has the following data structures (Fig. above):

- **PrQ** and **StallBuffer**: PrQ is a priority queue that contains activities in enabled state and local-stalled state. The StallBuffer contains activities in depend-stalled and remote-stalled states. The total size of both these data-structures together is O ($D_{max}.S_{max}$) bytes.
- **Ready Deque**: Also referred to as Deque, this contains activities in the current executing path on this worker. This has total space of O (S1) bytes.
- **AMRejectMap**: This is a one-to-one map from a place-id, say P_j, to the tuple [U, AM (V), *head*, *tail*]. This tuple has the following components:
  - AM(V): active message rejected in a remote-spawn attempt at place P_j
  - U: activity stalled due to the rejected active message
  - *head* and *tail* of the linked list of activities in *remote stalled* state due to lack of space on the place, P_j.

This map occupies O ($n.S_{max}$) space per worker.

Each place, P_i, has the following data-structures (Fig. above):

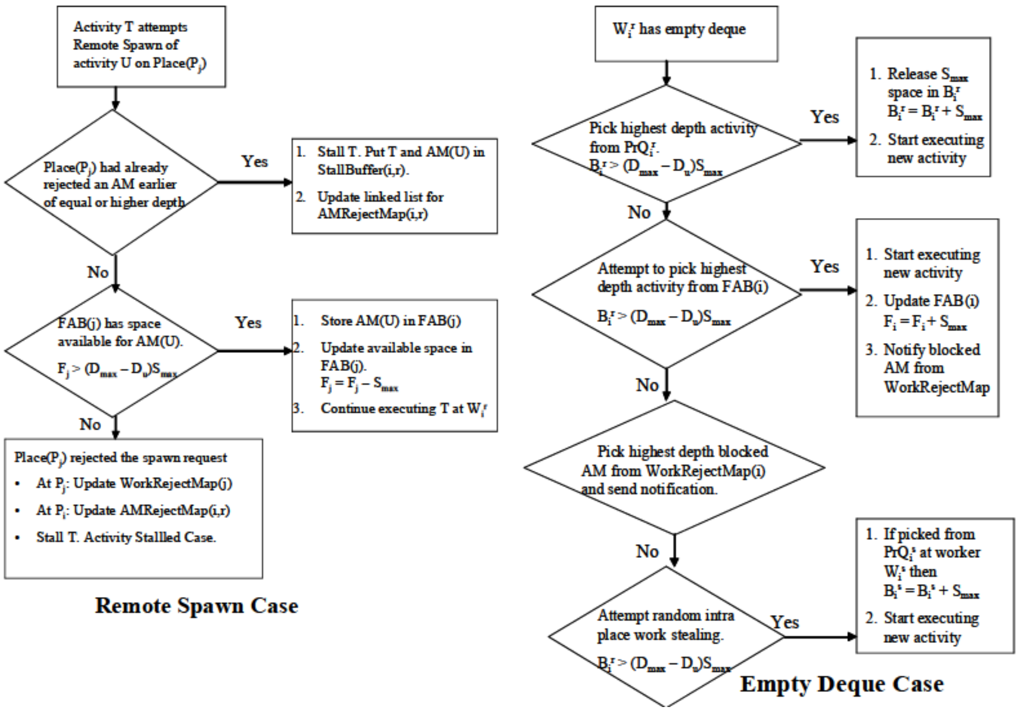
- **FAB**: This is a concurrent priority queue that is managed by a dedicated processor (different from workers). It contains the fresh activities spawned by remote places onto this place. It occupies O ($D_{max}.S_{max}$) bytes per place.
- **WorkRejectMap**: This is a one-to-many map from the computation depth to list of workers. For each depth, this map contains the list of workers whose spawns were rejected from this place. It occupies O (m.n + D_{max}) space.

The priority queue (used for PrQ and FAB) uses the depth of an activity as the priority with higher depth denoting higher priority.
The (computation) depth of an activity is defined as the distance from the root activity in the computation tree.
Consider the following notations.

- P = {P_1, …, P_n} denotes the set of places.
- {W_i^1, Wi2 ...W_i^m} denote the set of workers at place P_i.
- S_1 denotes the space required by a single processor execution schedule.
- S_{max} denotes the size in bytes of the largest activation frame in the computation.
- D_{max} denotes the maximum depth of the computation tree in terms of number of activities.

Further, let AMRejectMap (i, r), PrQ (i, r) and StallBuffer (i, r) denote the AMRejectMap, PrQ and StallBuffer respectively for worker W_r^i at place Pi. Let B_i^r denote the combined space for the PrQ (i, r) and StallBuffer (i, r). Let FAB (i) and WorkRejectMap (i) denote the FAB and WorkRejectMap respectively at place Pi. Let Fi denote the current space available in FAB (i). Let AM (T) denote the active message for spawning the activity T. The activities in remote stalled state are tracked with a linked list using activity IDs with the head and tail of the list available at the tuple corresponding to the place in map AMRejectMap.

Computation starts with root (depth 1) of the computation DAG at a worker W_s^0, at the default place P_0. At any point of time a worker, W_i^r, can either be executing an activity T, or be idle. When T needs to attempt a remote spawn at place P_j, it first checks if there are already stalled activities in AMRejectMap (i, r). If there is already a stalled activity, then T is added to the StallBuffer (i, r) and the link from the current tail in the tuple corresponding to P_j in AMRejectMap (i, r) is set to T. Also, the tail of the tuple is set to T. If there is no stalled activity in AMRejectMap (i, r) for place P_j, then the worker attempts a remote spawn at place P_j. At P_j check is performed by the dedicated processor for space availability in the FAB (j). If it has enough space then the active message, AM (U), is stored in the remote FAB (j), the available space in FAB (j) is updated and T continues execution. If there isn't enough space then AMRejectMap (i, r) is updated accordingly and T is put in the StallBuffer (i, r). When the worker W_i^r receives notification of available space from place P_j, then it gets the tuple for P_j from AMRejectMap (i, r) and sends the active message and the head activity to $P_{j.}$ At P_j, the WorkRejectMap (j) is updated. Also, W_i^r updates the tuple for P_j by updating the links for the linked list in that tuple. The remote stalled activity is enabled and put in PrQ (i, r). When currently executing activity T terminates, then the worker picks the bottommost activity to execute if one such exists. Else, the space reserved by T is released from B_i^r and the worker follows the same as the case for Empty Deque.


Activity T attempts Remote Spawn of activity U on Place(P_j)
Place(P_j) had already rejected an AM earlier of equal or higher depth
Yes
1. Stall T. Put T and AM(U) in StallBuffer(i,r).
2. Update linked list for AMRejectMap(i,r)
No
FAB(j) has space available for AM(U).
$F_j > (D_{max} - D_u)S_{max}$
Yes
1. Store AM(U) in FAB(j)
2. Update available space in FAB(j). $F_j = F_j - S_{max}$
3. Continue executing T at W_i^r
No
Place(P_j) rejected the spawn request
• At P_j: Update WorkRejectMap(j)
• At P_i: Update AMRejectMap(i,r)
• Stall T. Activity Stallled Case.
Remote Spawn Case
W_i^r has empty deque
Pick highest depth activity from PrQ_i^r. $B_i^r > (D_{max} - D_u)S_{max}$
Yes
1. Release S_{max} space in B_i^r $B_i^r = B_i^r + S_{max}$
2. Start executing new activity
No
Attempt to pick highest depth activity from FAB(i)
$B_i^r > (D_{max} - D_u)S_{max}$
Yes
1. Start executing new activity
2. Update FAB(i) $F_i = F_i + S_{max}$
3. Notify blocked AM from WorkRejectMap
No
Pick highest depth blocked AM from WorkRejectMap(i) and send notification.
No
Attempt random intra place work stealing.
$B_i^r > (D_{max} - D_u)S_{max}$
Yes
1. If picked from PrQ_i^s at worker W_i^s then $B_i^s = B_i^s + S_{max}$
2. Start executing new activity
Empty Deque Case

When the Deque becomes empty then the worker attempts to pick the available activity of maximum depth from its PrQ. If that does not succeed, then it tries to pick the available activity of maximum depth from FAB (i). If that also fails then it looks for any entries in the map WorkRejectMap and sends notification to the appropriate worker whose activity it can execute. If this also fails then the worker tries to randomly steal an activity of appropriate depth from another worker at the same place. When an activity T stalls then its state is set to appropriate stalled state and it is removed from Deque and put in either PrQ or StallBuffer depending on whether it is in local stalled state or not. The next bottommost activity U is picked from the Deque. If there is enough space to execute this activity then it is picked for execution else it is also stalled.

Conclusion & Future Work

PGAS offer great features for task and data parallelization. Different PGAS favors parallelization of different type in different application. We can make better use of our computing facilities by using the most appropriate for our application.

Parallel Breadth First search (PBFS) and Parallel Depth First (PDFS) search has been implemented for single place SMP architecture.

<u>Future Work for PBFS</u>

- ❖ Testing and benchmarking the Shared Memory Parallelized Implementation on SMP architectures (using one place).
- ❖ Improving the current implementation to introduce dynamic work distribution scheme between activities.
- ❖ Extending the similar kind of approach to implement an Efficient Distributed Memory Parallelized version for Distributed Memory architectures(Clusters etc.) and on SMP architectures(using more than one place)

<u>Future Work for PDFS</u>

- ❖ Testing and benchmarking the Shared Memory Parallelized Implementation on SMP architectures (using one place).
- ❖ Extending the similar kind of approach to implement an Efficient Distributed Memory Parallelized version for Distributed Memory architectures(Clusters etc.) and on SMP architectures(using more than one place)

We have addressed the challenging problem of affinity driven, online, deadlock-free distributed scheduling for parallel computations. This is the first such work for affinity driven distributed scheduling of parallel computations in a multi-place setup.

<u>Future Work</u>

- ❖ Testing and benchmarking the current scheduling algorithm on cluster of SMP machines.
- ❖ Comparing the results found out with the existing scheduling kernels targeting cluster of SMP

machines (MPI+Pthreads Implementation, KAAPI).

- ❖ Look into Markov-Chain based modeling and multicore cluster (such as Blue Gene/P) based implementation of our distributed scheduling algorithm.

Bibliography & References

1. Rao, V.N., Kumar, Vipin: Parallel depth first search. Part I. Implementation. International Journal of Parallel Programming, Volume 16 Issue 6, December 1987
2. Katherine Yelick and Dan Bonachea et.al. Productivity and performance using partitioned global address space languages. In PASCO ’07: Proceedings of the 2007 international workshop on parallel symbolic computation, pages 24–32, New York, NY, USA, 2007. ACM.
3. Agarwal, S., Narang, A., Shyamasundar, R.K.: Affinity driven distributed scheduling algorithms for parallel computations. Tech. Rep. RI09010, IBM India Research Labs, New Delhi
4. Leiserson, Charles E., Schardl, Tao B.: A work-efficient parallel breadth-first search (or how to cope with the non-determinism of reducers. In SPAA ’10 Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures.
5. S. Agarwal, R.Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yellick. Deadlock-free scheduling of x10 computations with bounded resources. In SPAA, pages 229 – 240, San Diego, CA, USA, December 2007.
6. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kermal Ebcioglu, Christoph von Praun, Vivek Sarkar: X10: an object-oriented approach to non-uniform cluster computing. In OOPSLA ’05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications
7. M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In SPAA, pp. 79–90, 2009.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. The MIT Press, third edition, 2009.
9. Supertech Research Group, MIT/LCS. Cilk 5.4.6 Reference Manual, 1998. Available from http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf
10. http://www.pgas.org/
11. http://chapel.cray.com/
12. http://software.intel.com/en-us/articles/intel-cilk-plus/
13. http://en.wikipedia.org/wiki/Intel_Cilk_Plus

14. http://supertech.csail.mit.edu/cilk/
15. http://x10plus.cloudaccess.net/home
16. http://x10.codehaus.org/
17. http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf
18. http://www.open-mpi.org/
19. https://computing.llnl.gov/tutorials/mpi/