An Adaptive Framework for Large-scale State Space Search

Yanhua Sun, Gengbin Zheng, Pritish Jetley, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{*sun51, gzheng, pjetley2, kale*}*@illinois.edu*

***Abstract*—**
State space search problems abound in the artificial intelligence, planning and optimization literature. Solving such problems is generally NP-hard. Therefore, a brute-force approach to state space search must be employed. It is instructive to solve them on large parallel machines with significant computational power. However, writing efficient and scalable parallel programs has traditionally been a challenging undertaking. In this paper, we analyze several performance characteristics common to all parallel state space search applications. In particular, we focus on the issues of grain size, the prioritized execution of tasks and the balancing of load among processors in the system. We demonstrate the techniques that are used to scale such applications to large scale. We have incorporated these techniques into a general search engine framework that is designed to solve a broad class of state space search problems. We demonstrate the efficiency and scalability of our design using three example applications, and present scaling results up to 16,384 processors.

***Keywords*-Parallel state space search, adaptive grain size control, dynamic load balancing, prioritized execution**

I. Introduction

State space search problems such as N-Queens [1], traveling salesman problem and various scheduling problems are commonly used in the field of operations research and artificial intelligence. Given that no polynomial-time algorithms are known to exist, these problems are solved through a systematic exploration of all possible configurations of their inherent elements. Each such configuration is termed a *state*, and the set of all possible configurations is called a *state space*. Generally, an operator is available to transform one state into another through the modification of the former's configuration. The objective of the state space search problem is to find a path from a *start* state to a desired *goal* state (or a path from the start to each among a set of goal states). Note that the problem is different from a regular graph search, since for most instances of interest the problem size prohibits the explicit enumeration of all the states and edges in before the search procedure begins. Most often, the search is seen to be tree-based – at each step of the search, we transform a stored *parent* state into several *children* states that do not violate the constraints specified by the problem. In this sense, we also refer to states as *nodes* of a search tree.

Several real-world applications use the state space search technique, for example floor-plan design in VLSI, genetic search, game-playing programs such as chess solvers, etc. As mentioned previously, most such problems are NP complete, so typically the time to solution increases exponentially with the problem size. Due to the high computational requirements and inherent parallel nature of the search techniques, there has been a great deal of interest in developing parallel search methods [2]. The combination of improvements in hardware and algorithmic enhancements such as load balancing [3] will allow us to scale up and solve previously intractable problem instances. However, programming today's large HPC clusters and supercomputers based on multi-core chips is a tremendous challenge. Developers need to consider various algorithmic and performance issues such as different choices of search procedure, heuristic algorithms, grain size control and load balancing.

In this paper, we present a framework for *Par*allel *S*tate *S*pace *S*earch *E*ngine called *ParSSSE* that lightens the burden of programmers interested in developing state space search applications by obviating the need to write parallel code. It provides an abstract and extensible interface to the programmer, so that it has wide applicability. ParSSSE ensures good performance and scaling for applications by performing dynamic optimizations along several dimensions. These are identified in the paper through a study of three well known applications, each representing a different class of state space search problems. In particular, we investigate performance issues such as grain size, speculative computation, and load balancing. We also present performance and scaling results from applications written in the framework, thereby demonstrating its scalability.

The framework we describe in this paper supports all three modes of state space search, namely *all*-solution, *first*-solution and *optimal*-solution search. The objective of all-solutions search is to find *all feasible* configurations of the search space, of which N-Queens is a classic example. In a first-solution search, the goal is to identify *any feasible* configuration among an exponential number of possibilities, as is done in the 3-SAT problem. Optimal-solution search requires that the procedure report a solution which is *better* than all others with respect to some metric. An example of this paradigm would be the traveling salesman problem.

For optimal-solution search, the IDA* [4] and branch-and-bound [5] techniques are currently supported by *ParSSSE*. The framework can be extended with relatively little effort to support other forms of search, such as bidirectional, game-tree and AND-OR tree search.

II. CHARM++

We begin with a brief description of CHARM++, which is the parallel infrastructure on which ParSSSE is based. We also describe the execution model of the framework, so as to better motivate the discussions on performance issues that follow.

CHARM++ [6] is a machine independent parallel programming language that runs on most shared and distributed memory machines. It employs an object-oriented approach to parallel programming. The programmer decomposes the problem into collections of objects that embody its natural elements. These objects are migratable and message-driven. Their number is independent of, but typically much larger than, the number of physical processors used to run the application. This over-subscription of processors is termed *object-based virtualization*. The migratable objects are assigned to processors by the underlying adaptive runtime system. An object communicates with another by asynchronously invoking an *entry method* on it. Together, asynchronous messaging and object-based virtualization enable the dynamic overlap of communication and computation: a processor may overlap messaging latency with not just the sending object's succeeding computation, but also with useful computation of other objects on that processor. Previous work [7], [8], [9] has been done on top of CHARM++ to study the load balancing issues in state space search problems. Below, we describe some of the key features of the CHARM++ model that pertain to the search engine.

Each processor maintains a queue of messages to be delivered to CHARM++ objects placed within it. This is called the *incoming queue.* There is a corresponding *outgoing queue* that holds messages generated by the processor's objects. The main control loop can be described simply: a scheduler picks a message from the incoming queue and a method on an object, both of which are specified by the picked message. The method executes to completion and is non-preemptive.

In our implementation of the search engine, each object is a task t with an associated tree node n_t. Each t maintains a LIFO *node queue* with which it performs a local depth-first search under node n_t. At each step of the depth-first search, t pops a node n_{top} from the node queue and checks it for feasibility. If n_{top} is found to be feasible, a solution is reported. If not, n_{top} may be expanded to yield children nodes. Depending on the depth of n_{top}, and the amount of work done by t thus far (*cf.* § III-A) the children of n_{top} may be assigned to newly created CHARM++ objects. In this case, the search engine enqueues a *seed message* for each child in the outgoing queue of the processor. Each of these messages results in the creation of a new task object.

A new task may be placed on a different processor from the one that created it. This placement decision is made by the distributed seed load balancer (*cf.* § III-C). Furthermore, the seed message may have a priority bit-vector associated with it (*cf.* § III-B). This is accounted for by placing the seed message at the correct position within the recipient processor's incoming queue.

Finally, the quiescence detection framework of CHARM++ can be exploited to detect termination of the state space search process. It triggers exit when it determines that (a) all processors are idle, (b) all processors have exhausted their incoming queues, (c) no messages remain in the outgoing queue of any processor and (d) there are no messages in flight.

III. EFFICIENT PARALLELIZATION OF STATE SPACE SEARCH PROBLEMS

As described previously, we adopt a task-based parallelization scheme, wherein each task is responsible for the exploration of a subset of the state space. These tasks are short-lived, but may spawn new tasks. Initially, newly created tasks are placed into a local task pool on the creating processor. Tasks are then distributed to other processors for load balancing. There are several issues associated with the efficient execution of spawned tasks in this manner. We describe some of these challenges next, and present techniques incorporated into ParSSSE to overcome them.

A. Adaptive grain size control

A key consideration in the design of a parallel search application is the parallelization strategy. One must decompose the search space into tasks that creates enough parallelism, while keeping the overheads of task creation and scheduling to a minimum.

State space search applications have two characteristic phases, namely startup and saturation. The startup phase begins with the expansion of the root or initial state into its children. During this phase the goal of an efficient parallel search procedure is to quickly generate enough work to saturate the processors available. Therefore, at this stage, a fine-grained decomposition of tasks is required. Once there is enough work for all processors to do (recall that tasks are short-lived, but generate child tasks, which are spread across the parallel machine using a load balancing technique), we enter the saturation phase. The goal here is to minimize the amount of overhead incurred in performing the parallel search. Therefore, each task must be of medium grain size, so that the overheads of creating and scheduling it are small in comparison to the amount of sequential computation it performs. Once the task has performed the requisite amount of work, it may be allowed to spawn new children tasks, which are executed in parallel. Since most of the application

time is spent in the sequential search on the leaves, it is crucial to control their workload for purposes of load balance. We refer to this problem as a grain size control problem. Next, we propose a heuristic that adaptively adjusts the grain size.

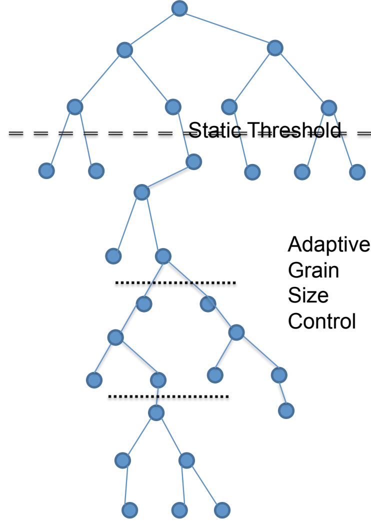

Figure 1. Adaptive grain size control

ParSSSE adopts a three-level grain size control strategy. At the most top level, the finest grain size is used for decomposition, i.e. each generated tree node is processed as an independent task. In this way, newly generated parallel tasks of increasing depth are spread across the processors as quickly as possible. If the depth of a scheduled task exceeds a particular threshold τ_1, the tree beneath its corresponding node is explored sequentially, in a depth-first manner using a LIFO queue. We choose a threshold of $\tau_1 = \log_b p$, where b is the branching factor and p is the number of processors. However, this static grain size control is far from effective, especially in the case of unbalanced tree search, where sequential tasks have widely varying execution times, thereby creating load imbalance. To overcome this problem, and to guarantee that no sequential execution is extraordinarily long, nodes present in the LIFO queue of the sequential search are parallelized by generating separate parallel tasks. Figure 1 shows the use of this strategy for an unbalanced search tree.

The key to a good adaptive grain size control scheme is the decision of when the LIFO queue of a task is split and new tasks are launched.

Figure 2 shows a schematic depiction of the relationship between the average grain size of parallel tasks and the obtained speedup. Marked on this graph are two values for grain size, g_0 and g_1, that occur on either side of the optimal value. If an application generates tasks with average grain size less than g_0, it suffers from large overheads of parallelization. On the other hand, employing tasks with

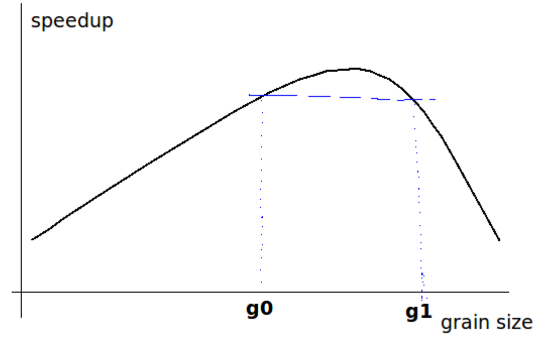

Figure 2. Grain size vs. speedup

average grain size greater than g_1 reduces the utilization of processors, leading to a slowdown. In order to achieve a reasonable speedup, the grain size should be controlled between the g_0 and g_1. Assuming that the overhead of creation and scheduling of a parallel task is a constant t_o, and the average grain size of a parallel task is t_g, we estimate the total time taken to complete an all-solutions search in parallel on p processors to be:

$$
\begin{aligned}
T_p &= \frac{T_{seq}}{p} + T_{overhead} + T_{idle} \\
&= \frac{T_{seq}}{p} + \frac{t_o}{t_g} \cdot T_{seq} + t_g \cdot (n_{max} - \frac{T_{seq}}{p \cdot t_g})
\end{aligned}
$$

Here, n_{max} is the number of parallel tasks executed by the processor which received the greatest number of tasks during the execution, and T_{seq} is the time taken to complete the search sequentially. One straightforward solution to control the grain size is to measure the time expanding each subtree. When the measured time is greater than some pre-defined threshold, the subtree is split into multiple parallel tasks to maintain enough parallelism. However, in practice, the cost of timer calls used to measure the expanding of a subtree is relatively high comparing to the time expanding the tree node. To reduce the timer cost, we propose a sampling solution. When exploring a particular subtree sequentially, due to the limitation of the branching factor, the amount of time expanding a tree node into multiple children is roughly the same for this particular subtree. Therefore, by sampling the time taken to expand a few nodes, we calculate the estimated average time to expand a single tree node. The total time of expanding a subtree can be estimated by extrapolating to the actual total number of nodes expanded. If this calculated time (as grain size) is more than ten times of t_o, the sequential task's LIFO is split and k new tasks are created. The value of k may be decided by the user.

B. Speculation and Prioritized Execution

Performing a search in parallel involves some degree of speculative work, since nodes that might not have been considered in the sequential search may be explored in a parallel search of the state space. Rao *et al.* have discussed the presence effects of this speculation, leading to superlinear as well as sublinear speedups as the number of processors is scaled up. The amount of speculative computation performed in the distributed depth first search can be controlled by associating a priority with the execution of each parallel task [9], [10]. The value of this priority corresponds to the lexicographical labeling of nodes in the tree. Tasks are then executed in the ascending order of priorities. This scheme has the following benefits:

Reduced memory footprint. For an all-solutions search, prioritized execution helps to reduce memory usage in the following manner. A task corresponding to the leftmost child of a node is given priority over its siblings. Furthermore, all the descendants of such a left child are given priority over its siblings. Therefore, the search can be made to resemble a depth-first exploration of the state space. This limits the size of the search frontier to $O(bd)$. Without prioritization of tasks, the search would be closer to a breadth-first exploration with $O(b^d)$ memory requirements.

Reduced speculation. For a heuristic-guided first-solution search, the usage of priority is even more important. Each task is assigned a priority that corresponds to the likelihood of finding a solution under its node, as determined by the heuristic function. Thus, prioritization of tasks can guide the search in the direction of nodes that are more likely to yield solutions, thereby reducing the amount of speculative computation performed.

To keep our search engine framework general, two types of priorities are supported, namely bitvector priorities and integer priorities. Integer priorities are useful when there is an evaluation function according to which the nodes of the search tree are ordered. The value of the evaluation function can be converted into an integer priority, and attached to the parallel task created.

Bit-vector priorities are somewhat more complicated. These are bit-strings of arbitrary length that represent fixed-point numbers in the range 0 to 1. For example, the bit-string “001001” represents the number $.001001_{\text{binary}}$. As with integer priorities, higher numbers represent lower priorities. Bitvector priorities are especially useful when there is no explicit evaluation function that can be used to assign integer priorities. The root of the search tree is assigned a single bit priority string, 0. The bitvector priority of a child is obtained by concatenating the binary representation of the child’s rank to the bitvector of the parent. For example, for a binary tree, the binary ranks for the two children of any node would be 0 and 1. For a quad-tree, the ranks would be 00, 01, 10, and 11. The use of bitvector priorities in a binary tree is illustrated in Figure 3.

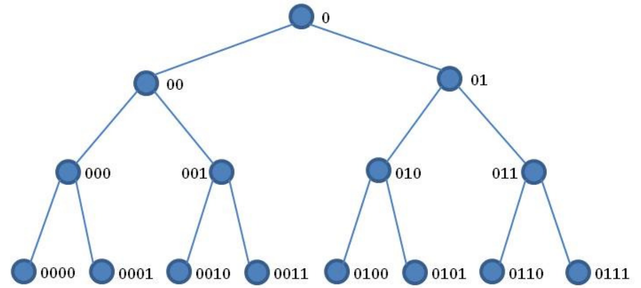

Figure 3. Bitvector priorities in state space search

C. Distributed Task scheduling

In order to dynamically achieve a good balance of load among the processors in a parallel setting, created tasks need to be distributed evenly across all processors. This work-pool based load balancing problem has been well studied in the literature [11], [12]. Given the large variety of load balancing algorithms, each may work well for certain scenarios, users are left facing a challenging problem of either choosing the best load balancing strategies or writing better ones. In ParSSSE, we focus on a modular design that allows a clean separation of the load balancing procedure from the user code, which makes it easy to develop and plug in any load balancing algorithms in Charm++. The evaluation of different load balancing strategies is not in the scope of this paper.

We investigate the design of a general load balancing framework that allows fully distributed load balancing strategies to be plugged in at run-time. This module defines a core task dispersal function called *CldEnqueue*, that initiates a new task by sending a message contatining it to a lightly-loaded processor.

When a new task is created by the program on a processor, the dispersal function places its corresponding message into a pool of movable work. This pool gradually shrinks as it is processed, but in most programs we observe a sufficient amount of movable work at any given time. As load conditions change, the load balancers move the work in the pool around. A message may be shifted more than once, with the number of moves being limited by an upper bound. This allows work to be shifted to the optimal lightly loaded processor, while ensuring that the overhead of doing so remains bounded. Processors exchange work from each others’ pools periodically, and also when their own queues call below a certain threshold. More proactive strategies may be employed, so that newly generated work is sent to the neighborhood of a processor, as decided by a virtual processor topology.

Priorities are also accounted for. In addition to ensuring that all processors have some work to do, the dispersal

function assigns work such that every processor has some reasonably high-priority work to do. This feature is of great utility in first-solution, optimal-solution and branch-and-bound searches, since high priority work corresponds to regions of the search tree that are considered more likely to yield solutions, thereby reducing the amount of speculation.

Several load balancing strategies are available, and new ones may be written by users. In addition, different virtual topologies, such as dense trees and k-dimensional torii, may be used for processors, making this a flexible and extensible framework.

IV. Application Programming Interface

In this section, we familiarize the reader with some of the specifics of the application interface presented by the search engine.

The code below shows the skeleton of the important parts of the search engine framework. It also presents the handful of functions that need to be implemented by the user in order to obtain a working parallel search program.

```
class BTreeState : public StateBase{
public:
 int depth;
 /*user defined data structure*/
};

void createInitialChildren(Solver *solver)
{
  BTreeState *root=(BTreeState*)solver->
                       registerRootState(
                       sizeof(BTreeState),
                       0,
                       1);
  root->depth = 0;
  solver->process(root);
}

inline void createChildren(StateBase *_base,
                          Solver *solver,
                          bool parallel)
{
  BTreeState &base = *((BTreeState*)_base);
  for(int childIndex=0;
          childIndex<branchfactor;
          childIndex++)
  {
    if(base.depth == depth-1)
      solver->reportSolution();
    else{
      BTreeState *child=(BTreeState*)solver->
                     registerState(
                     sizeof(BTreeState),
                     childIndex,
                     branchfactor);
      child->depth = base.depth + 1;
      if(parallel) solver->process(child);
    }
  }
}

int parallelLevel()
{ return initial_grainsize; }

int searchDepthLimit()
{ return 1; }

SE_Register(BTreeState,
            createInitialChildren,
            createChildren,
            parallelLevel,
            searchDepthLimit);
```

The *BTreeState* class must be inherited from the StateBase class. It encapsulates all the data and operation of the search state. The *createInitialChildren(..)* function creates the root node(s) of the search tree. The *createChildren(..)* function takes the parent node as input and creates new children nodes. The *parallelLevel()* function returns the static parallel search depth threshold, which is discussed in section III-A. The *searchDepthLimit()* function returns the initial maximum search depth if it is an Iterative Deepending A* (IDA*) search problem. The *lowerBound(..)* functions is required in branch-and-bound search problem. It returns the lower bound value for a given state. *registerState(..)* allocates memory for a new state which is the mth child of the total n children. Users do not have to explicitly allocate or manage memory, which is all in control of search engine. Function *process(..)* is called after the data of the state is filled and ready to be executed.

Figure 4 illustrates the functions in layers of user program, ParSSSEand CHARM++. From this figure, we can see that the user code devoids of any details of parallel execution. All the issues discussed in section III are handled by the search engine.

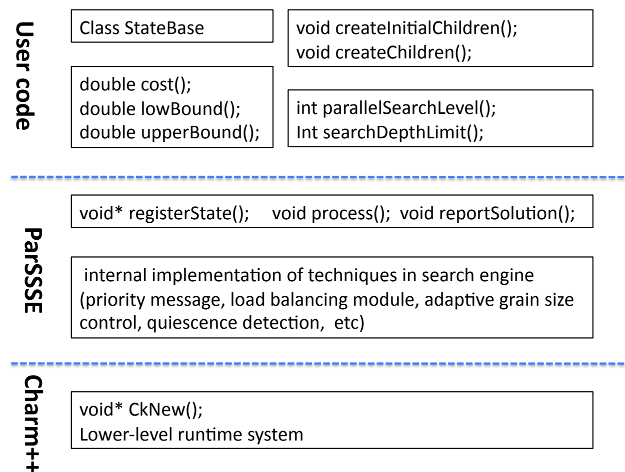

Figure 4. ParSSSE Design and Implementation

V. Experimental Results

This section presents several applications that we have used to benchmark the performance and scalability of our search engine. All the experiments are run on Intrepid, a Blue Gene/P installation at Argonne National Laboratory. Intrepid has 40 racks, each of them containing 1024 compute nodes. A node consists of four PowerPC450 cores running at 850 MHz. Each node has 2 GB of memory.

A. Benchmarks

1) Balanced search tree (BST): This problem searches an abstract complete binary tree in which some leaf nodes are solutions. This program is used to illustrate the super-linear and sub-linear speedup in parallel in paper [13].

In this problem, we assume that the first solution leaf node we want to find is the m_{th} leaf in the deepest level. For convenience, we suppose that m is the power of 2.

2) NQueens: NQueens is a backtracking search problem to place N queens on a N by N chess board so that they do not attack each other. We target at finding all solutions for N Queen problem.

3) Unbalanced tree search (UTS): This is a parallel exploration of an unbalanced search tree[14]. Its performance greatly depends on how well the adaptive grain size control and load balancing works. In our experiments, we take two instances, which are *T1XXL* and *T3XXL*. The parameters [14] for them are in table I.

Instance	t	a	d	b	r
T1XXL	1	3	15	4	19
T3XXL	0	2000	0.499995	2	318

Table I
UNBALANCED TREE SEARCH INSTANCES - T1XXL, T3XXL

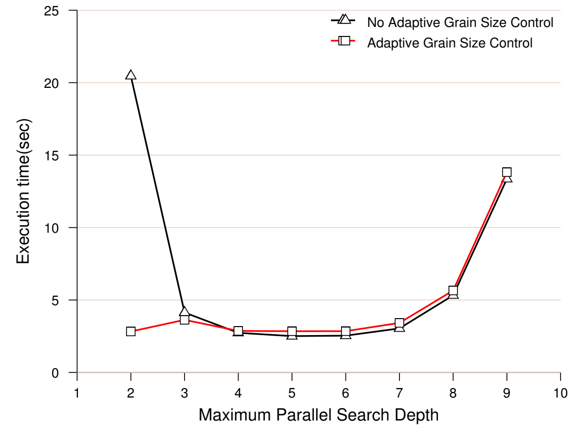

Figure 5. 18-Queen execution time on 1,024 processors

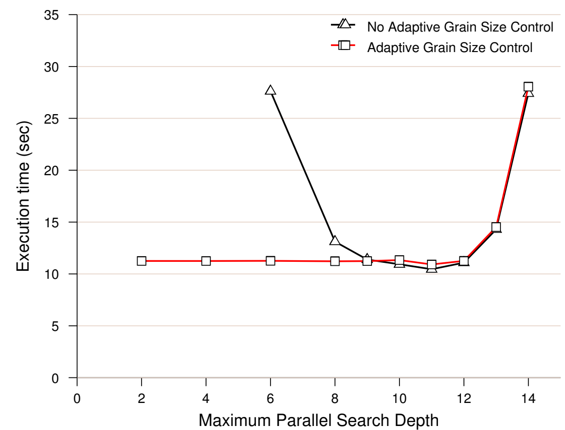

Figure 6. T1XXL execution time on 1,024 processors

B. Experiments

Adaptive Grain Size Performance: In order to test how the adaptive grain size control helps the parallel program performance, we ran the *18 Queens* and *T1XXL, T3XXL UTS* instances with different initial grain size and compared the static grain size control with the adaptive method.

Figure 5 shows the execution time for *18 Queens* on 1024 processors with adaptive grain size control compared with only static grain size threshold. It shows for an initial coarse grain size threshold, adaptive grain size control performs much better than the one with only static threshold. Although the best performance with adaptive grain size control is a little worse (10%), the performance variance with adaptive grain size control is much smaller. We also observe that with finer grain size, both perform bad because of the high overhead. This suggests that it is a good strategy to first choose a coarse-grain threshold and then use the adaptive grain size control to fine-tune the performance. Same observation is found for T1XXL test in figure 6.

Prioritized Execution Tests: We used the *BST* to test how the prioritized execution helps reduce the number of speculative nodes processed in search engine framework. In our experiments, the tree depth is set as 30. Thus the total number of leaves in the tree is 2^{30}. Our search target is to find the first node in the deepest level. This experiment simulates the process of searching a tree with heuristic in programs, where solutions are more likely in the left branch subtrees. From figure 7, we can see that priority execution helps reduce a huge number of nodes created and processed comparing with no prioritized execution. The reason why the difference becomes smaller with processor number increasing is that the number of tree nodes on each processor decreases so that finding a solution on one processor prevents further nodes creation and processing on all other processors.

Scalability: We have tested the scalability of our NQueens and UTS implementation based on search engine from 512

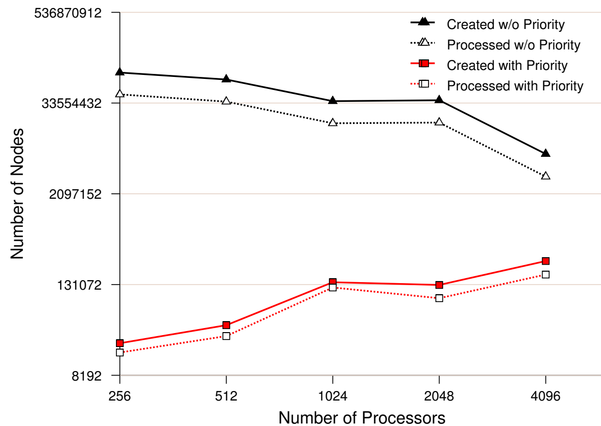

Figure 7. Number of nodes created and processed in BST w/o priority execution

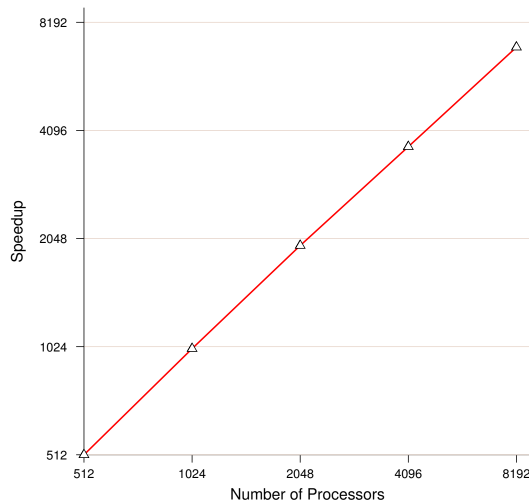

Figure 8. 18-Queen speed-up from 512 processors to 8,192 processors

processors up to 8,192 and 16,384 processors respectively. The results for *18 Queens* with initial search depth 5 is shown in figure 8. We have achieved 85.25% efficiency on 8,192 processors comparing to 512 processors. For T1XXL instance in figure 9, we have achieved even better performance because of the higher ratio of computation to communication in UTS. Based on the performance of 512 processors, the efficiency on 8,192 processors is 98.75% while it is 88.28% on 16,384 processors.

VI. Related Work

Much work has been done on parallel combinatorial search in the past. Good surveys of the field have been provided by Grama and Kumar [15] and Nelson and Toptsis [16] and, in the context of parallel logic programming, by Gupta *et al.* [17]. Particular paradigms in parallel combinatorial

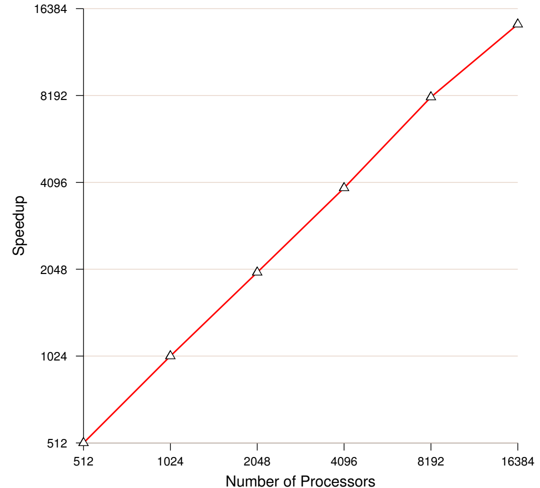

Figure 9. T1XXL speed-up from 512 processors to 16,384 processors

search have also been explored extensively – see [18], [19] for discussions of techniques and heuristics in branch-and-bound, [4] for parallel iterative deepening A*, [20] for game-tree search and [21] for parallel search of AND-OR trees. The importance of cutoff-based strategies in task parallel languages is highlighted in [22]. A discussion of the impacts of granularity on the performance of OR-parallel programs has been given by Furuichi *et al.* [23]. That paper also provides a multi-level load balancing scheme for multiprocessor systems. The use of priorities in a variety of parallel search contexts has been outlined by Kalé *et al.* [8]. The use of work-stealing as a load balancing strategy was first described by Lin and Kumar [24] and subsequently popularized by the Cilk system [2]. The work on Cilk also provides in-depth asymptotic bounds on the performance of the work-stealing approach.

VII. Conclusion

Solving state space search problems is generally NP-hard, and requires parallel processing to speed up the search process. However, writing efficient and scalable parallel programs has traditionally been a challenging undertaking. The main contribution of our paper is the design of the ParSSSE framework that separates the issues of parallelism and scalability from those of specifying the search tree itself. In this paper, we analyzed several performance characteristics common to all parallel state space search applications. In particular, we focused on the issues of grain size, prioritized execution of tasks and balancing of load among processors in the system. We show how these techniques may be used to scale such applications to large scale. We have incorporated these techniques into a general search engine framework ParSSSE, that is designed to solve a broad class of state space search problems. We demonstrated the

efficiency and scalability of our design using three example applications, presenting good performance results on up to 16,384 processors.

Acknowledgments

This work was supported in part by NSF grant OCI-0725070 for Blue Waters deployment and NSF grant ITR-HECURA-0833188, by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign and by the NIH grant PHS 5 P41 RR05969-04. We used running time on the Blue Gene/P at Argonne National Laboratory, which is supported by DOE under contract DE-AC02-06CH11357.

References

[1] L. Kalè, "An almost perfect heuristic or the N-queens problem," *Information Processing Letters*, vol. 34, no. 4, pp. 173–178, April 1990.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[3] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.

[4] V. N. Rao, V. Kumar, and K. Ramesh, "A parallel implementation of Iterative-Deepening-A*," in *AAAI*, 1987, pp. 178–182.

[5] R. M. Karp and Y. Zhang, "Randomized parallel algorithms for backtrack search and branch-and-bound computation," *J. ACM*, vol. 40, no. 3, pp. 765–789, 1993.

[6] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[7] V. Saletore and L. Kale, "Consistent linear speedups for a first solution in parallel state-space search," in *Proceedings of the AAAI*, August 1990, pp. 227–233.

[8] L. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha, "Prioritization in parallel symbolic computing," in *Lecture Notes in Computer Science*, T. Ito and R. Halstead, Eds., vol. 748. Springer-Verlag, 1993, pp. 12–41.

[9] A. Sinha and L. Kalé, "A load balancing strategy for prioritized execution of tasks," in *Seventh International Parallel Processing Symposium*, Newport Beach, CA., April 1993, pp. 230–237.

[10] L. Kalé and V. Saletore, "Parallel state-space search for a first solution with consistent linear speedups," *Internaltional Journal of Parallel Programming*, vol. 19, no. 4, pp. 251–293, 1990.

[11] L. V. Kalé, "Comparing the performance of two dynamic load distribution methods," in *Proceedings of the 1988 International Conference on Parallel Processing*, St. Charles, IL, August 1988, pp. 8–11.

[12] W. W. Shu and L. V. Kalé, "A dynamic load balancing strategy for the Chare Kernel system," in *Proceedings of Supercomputing '89*, November 1989, pp. 389–398.

[13] V. N. Rao and V. Kumar, "Superlinear speedup in parallel state-space search," in *Proceedings of the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK: Springer-Verlag, 1988, pp. 161–174.

[14] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in *Lecture Notes in Computer Sciences*, vol. 4382. Springer-Verlag, 2007, pp. 235–250.

[15] A. Grama and V. Kumar, "State of the art in parallel search techniques for discrete optimization problems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 28–35, 1999.

[16] P. C. Nelson and A. A. Toptsis, "Unidirectional and bidirectional search algorithms," *IEEE Software*, vol. 9, pp. 77–83, March 1992.

[17] G. Gupta, E. Pontelli, K. A. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: a survey," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 4, pp. 472–602, 2001.

[18] G.-J. Li and B. W. Wah, "Coping with anomalies in parallel branch-and-bound algorithms," *IEEE Transactions on Computing*, vol. 35, no. 6, pp. 568–573, 1986.

[19] T.-H. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *Commun. ACM*, vol. 27, pp. 594–602, June 1984. [Online]. Available: http://doi.acm.org/10.1145/358080.358103

[20] R. Feldmann, P. Mysliwiete, and B. Monien, "Studying overheads in massively parallel min/max-tree evaluation," in *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1994, pp. 94–103.

[21] L. Kalé, "The REDUCE OR process model for parallel execution of logic programs," *Journal of Logic Programming*, vol. 11, no. 1, pp. 55–84, July 1991.

[22] H.-W. Loidl and K. Hammond, "On the granularity of divide-and-conquer parallelism," in *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Jul. 1995.

[23] M. Furuichi, K. Taki, and N. Ichiyoshi, "A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi," in *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, pp. 50–59.

[24] Y.-J. Lin and V. Kumar, "And-parallel execution of logic programs on a shared-memory multiprocessor," *J. Log. Program.*, vol. 10, no. 1/2/3&4, pp. 155–178, 1991.