Parallel Processing Letters

ParSSSE: AN ADAPTIVE PARALLEL STATE SPACE SEARCH ENGINE

YANHUA SUN, GENGBIN ZHENG, PRITISH JETLEY, LAXMIKANT V. KALE
Department of Computer Science, University of Illinois at Urbana-Champaign
201 N Goodwin Ave, Urbana, Illinois, 61801, United States

Received June, 2011
Revised July, 2011
Communicated by Guest Editors

ABSTRACT

State space search problems abound in the artificial intelligence, planning and optimization literature. Solving such problems is generally NP-hard, so that a brute-force approach to state space search must be employed. Given the exponential amount of work that state space search problems entail, it is desirable to solve them on large parallel machines with significant computational power. In this paper, we analyze the parallel performance of several classes of 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 corresponding techniques that are used to scale such applications to large scale. Moreover, we tackle the problem of programmer productivity by incorporating these techniques into a general search engine framework 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 32,768 processors.

Keywords: parallel state space search, adaptive grain size control, dynamic load balancing, prioritized execution

1. Introduction

The state space search technique has varied applications. The traveling salesman problem and various scheduling problems are commonly encountered in the field of operations research and artificial intelligence. Other examples include floor-plan design in VLSI, genetic search for optimization, and game-playing programs such as chess solvers. Given that no polynomial-time algorithms are known to exist for these problems, they 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 *goal* state (or to each among a set of goal states). 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.

Although the basic mechanism employed in all state space search problems is essentially the same, namely the dynamic enumeration of elements of the state space, problem classes differ from each other in their requirements and the constraints they place on the search procedure. This leads us to categorize state space search problems in the following manner.

All solutions search. In this class of combinational search problem, the objective is to find all *feasible* configurations of the search space. The N-Queens problem is a classic example of this problem. A parallel exploration of the search tree is generally organized as a depth-first traversal, each processor charged with the exploration of a different portion of the tree. The various factors that determine the efficiency of this parallel search are discussed in Section 3.

First solution search. The objective of certain problems is to identify *any* feasible configuration among an exponential number of possibilities. The 3-SAT problem is exemplary of this class of search problems. The search procedure must find *an* assignment for a set of Boolean variables such that a given predicate is satisfiable. Any satisfying assignment suffices. A parallel depth-first procedure may be used to explore the state space. However, note the speculative nature of the computation: regions of the tree that may not be visited in a sequential procedure are searched concurrently in the hope of finding a solution more quickly. As documented in the literature, this can lead to the problem of *anomalous* speedups [23]. The addition of processors to the search may yield sublinear or superlinear speedups. This issue is discussed in more detail in Section 3.

Optimal solution search. An important class of problems requires that the search procedure report a solution which is *better* than all others with respect to some metric. Therefore, the objective of such a problem is to find an *optimal* solution. One could compute the optimal solution by considering each one found by a parallel all-solutions search, but better techniques have been presented in the literature. These are described below.

In many situations, it is possible to define a heuristic measure that provides, for each tree node n, a lower bound $l(n)$, which is the minimum path length from n to any solution within the subtree rooted at n. When considering an intermediate node n, such an *admissible* heuristic allows us to compute a lower bound $f(n)$ on the *total* cost of reaching a solution through n, $f(n) = l(n) + c(n)$, where $c(n)$ is the length of the path from the start state to n. It has been shown that if nodes are processed in ascending order of $f(n)$, the first solution found will be optimal [3]. This observation forms the basis for the A* search procedure. The best-first nature of the A* search procedure leads to memory requirements that are exponential in the problem size. When the solution cost is quantized (as is the case in the 15-puzzle) the Iterative

Deepening A* (IDA*) [24] technique may be applied. The procedure is arranged in a series of iterations. Starting with an initial bound d on the solution cost, in each iteration the state space is explored in a depth-first manner for solutions with cost no greater than d. If no such solution d is incremented and the search is restarted. The procedure is continued until a solution is found.

The branch-and-bound technique is a more general procedure used to find optimal solutions to state space search problems. As with IDA*, it uses a heuristic to rank nodes according to their estimated closeness to a solution. In this case, the heuristic must be both admissible and *monotonic*, i.e. the cost of a child node can be no smaller than that of its parent. The cost of the best known solution, B, is maintained across processors. This allows the procedure to prune the tree at nodes with greater cost than B. Whenever a solution s with a better cost than B is found, B is updated to the cost of s. Such updates are exchanged between processors so that the search does as little wasted work as possible.

Due to the high computational requirements and inherently parallel nature of such tree-based search techniques, there has been a great deal of interest in developing parallel methods [2, 7] for such search problems. Whereas most work in this context has been done more than twenty years ago, the combination of recent improvements in hardware and dynamic, runtime enhancements such as load balancing [4] allow us to scale to unprecedented levels, and consider much larger problems. However, programming today's large HPC clusters and supercomputers based on multicore 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. Moreover, these considerations must be addressed separately for each class of parallel state space search problems.

In this paper, we address the questions of performance and productivity in the context of parallel state space search methods. We present the *Par*allel *S*tate *S*pace *S*earch *E*ngine (*ParSSSE*), a framework that lightens the burden of programmers in developing state space search applications by obviating the need to write parallel code. It provides an abstract and extensible interface to the programmer. ParSSSE ensures good performance by performing dynamic optimizations along several dimensions. In particular, we investigate performance issues such as grain size, speculative computation, and load balancing. Corresponding optimization techniques are developed and integrated into *ParSSSE*. Results of three well-known applications written in the framework are presented to demonstrate 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. In particular, for optimal-solution search, the IDA* [24] and branch-and-bound [15] 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.

2. Charm++

We begin with a brief description of CHARM++, which is the parallel infrastructure on which ParSSSE is based. Next, we describe the execution model of the framework, which motivates the discussions on performance issues that follow.

CHARM++ [12, 10] is a machine independent parallel programming language with an accompanying runtime system that has been ported to 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 message-driven and migratable. 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. Asynchronous messaging and object-based virtualization enable the dynamic overlap of communication and computation: a processor may overlap messaging latency with not only the object's succeeding computation, but also with useful computation of other objects on the same processor. Some previous work [25, 13, 27] has used CHARM++ to study load balancing techniques in state space search problems. Below, we describe some of the key features of the CHARM++ model that pertain to the search engine.

In CHARM++, each processor maintains a queue of messages to be delivered to CHARM++ objects placed on that processor. This is called the *incoming queue.* There is also a corresponding *outgoing queue* that holds messages generated by the objects to other processors. On the processor, a scheduler keeps checking the incoming queue. When a message is available, the scheduler picks the message and invokes the method associated with the message to the corresponding object.

In our implementation of the search engine, each object represents 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. In the depth-first search, at each step, t picks a node n_{top} from the head of 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.* § 3.1), the children of n_{top} may be assigned to newly created tasks. 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.* § 3.3). Furthermore, the seed message may have a priority bit vector associated with it (*cf.* § 3.2). This is accounted for by placing the seed message at the correct position within the recipient processor's incoming queue.

Termination. For single-solution problems, the program must be terminated as

quickly as possible after the solution is found. This may require an *immediate* message, which skips the CHARM++ scheduler and interrupts the execution of the task currently occupying the processor. This ensures that the program does not perform unneeded computation. For all-solutions problems, the *quiescence detection* algorithm implemented in CHARM++ is exploited to detect the termination when all tasks are finished. The quiescence state is reached when (a) all processors have processed all messages in their incoming queues, (b) no messages remain in the outgoing queue of any processor, and (c) there are no messages in flight, i.e., the number of generated messages is equal to the number of messages processed.

Solution collection. In solving all-solutions problems on distributed-memory systems, care must be taken to reduce the time spent in collecting all the solutions. To minimize the impact of solution collection on the search procedure itself, ParSSSE collects solutions locally during the computation, instead of reporting them to processor 0 as they are found. When quiescence is detected, a spanning-tree based reduction operation is performed to collect all the solutions to processor 0.

3. Efficient Parallelization of State Space Search Problems

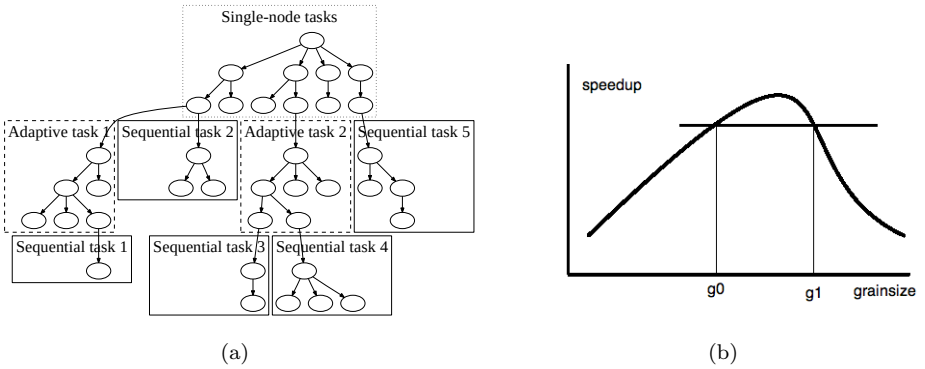
We now consider various aspects of parallel performance in a task-based parallelization scheme, wherein each task is responsible for the exploration of a subset of the state space. Tasks are short-lived, but may spawn new tasks. Depending on the load balancing strategies, newly created tasks are either placed into a local task pool on the creating processor or sent to the other processors. During execution, tasks are 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.

3.1. *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 so as to create enough parallelism, while keeping the overheads of task creation and scheduling to a minimum.

We consider these overheads in the context of the two characteristic phases of state space search algorithms, 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 (since 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. In this phase, the goal 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.



(a) (b)

Fig. 1. Adaptive grain size control. In (a) the following are shown: (i) shallower tasks comprise single search nodes, (ii) tasks are dynamically created to adapt to imbalances in search depth and (iii) parallel overhead is amortized over multiple nodes in sequential tasks; (b) shows schematically the expected effect of grain size on parallel speedup.

ParSSSE adopts a three-level grain size control strategy. At the topmost level, the finest grain size is used for decomposition in that 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 not very effective, especially in the case of unbalanced tree searches, where statically determined sequential tasks have widely varying execution time, thereby creating load imbalance. To overcome this problem, and to guarantee that no sequential execution is extraordinarily long, the unprocessed nodes in the LIFO queue of the sequential search are converted into parallel tasks when the sequential execution time exceeds a threshold. Figure 1(a) shows the use of this strategy for an unbalanced search tree.

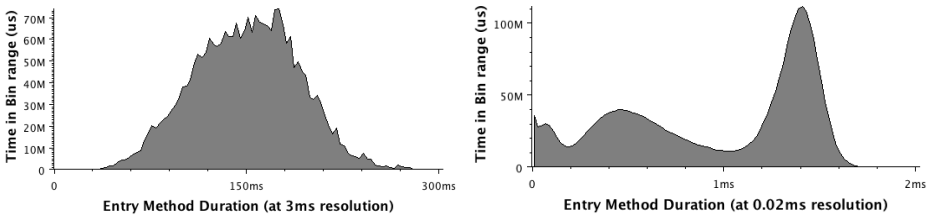
The key to a good adaptive grain size control scheme is the decision of when the LIFO queue of a task is split, causing new tasks to be generated. Figure 1(b) shows a schematic depiction of the relationship between the average grain size of

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, creating tasks with average grain size greater than g_1 reduces the utilization of processors, leading to a slowdown. In order to achieve a good speedup, the grain size should be controlled between 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:

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

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 spent in 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 use a solution based on sampling. When exploring a subtree sequentially, we assume that the branching factor at each node in a subtree is roughly the same. Therefore, by sampling the time taken to expand a few nodes, we can estimate the average time to expand a single tree node. The total time of expanding a subtree can then be estimated by extrapolating to the number of nodes expanded. If this calculated time (the grain size) is more than a pre-defined threshold, T, the sequential task's LIFO queue is split and k new tasks are created. We calculate T as follows. Let t_0 be the overhead of creating a parallel task. Then, since each task performs at least T units of sequential work before firing k children tasks, the average grain size in this case must be greater than or equal to $T/(k+1)$. At the same time, no task can take more than T units of time, so that the grain size can be no greater than T. Averaging over these two extremes, we heuristically set the average grain size to be $0.5 * (T/(k+1) + T)$. Assume that the desired sequential efficiency of the parallel program is $0 < e < 1$, i.e. the fraction of time taken up by parallel overhead is $1 - e$. In order to keep the impact of overhead negligible, the average grain size must be at least $\frac{t_0}{1-e}$, so that $T = \frac{2t_0(k+1)}{(1-e)(k+2)}$.

Figure 2 presents two histograms which compare the adaptive, sampling-based approach with the static threshold control scheme using an 18-queens problem. In each subfigure, the x-axis represents the duration of task entry methods (i.e. grain sizes) arranged into bins by certain resolutions, and y-axis depicts the total execution time for all the tasks in the corresponding bin. Figure 2(a) shows the result in the case of using the static grain size control scheme, where all nodes with



(a) Only with static threshold (b) With adaptive grain size control

Fig. 2. Accumulated execution time for different grain-sized parallel tasks (18-Queens problem)

depths greater than a pre-determined threshold were processed as sequential tasks. This scheme clearly creates a wide range of grain sizes, from a couple of microseconds to 300 milliseconds. In contrast, adaptive grain size control (Figure 2(b)) generates tasks that are less than 2 milliseconds long, which leads to better load balance and performance.

3.2. *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.* [23] have discussed the 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 [27, 14]. 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. Following the *delayed release* technique of [14], the search attains a characteristic *broomstick sweep* exploration of the state space. This limits the size of the search frontier to $O(bd+p)$. In contrast, if each processor were performing a depth-first traversal, the memory requirements of the program would be $O(bdp)$. Without any prioritization at all, the search would be resemble 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 the search engine framework general, two types of priorities are supported, namely bit vector 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. Bit vector priorities tend to create a left-to-right ordering of the tree, mimicking the sequential ordering. Local heuristic can still be used to order the children of a given node by assigning more promising children a lower rank. The root of the search tree is assigned a single bit priority string, 0. The bit vector priority of a child is obtained by concatenating the binary representation of the child's rank to the bit vector of the parent. As shown in Figure 3, the binary ranks for the two children of any node in a binary tree are 0 and 1. For a quad-tree, the ranks would be 00, 01, 10, and 11.

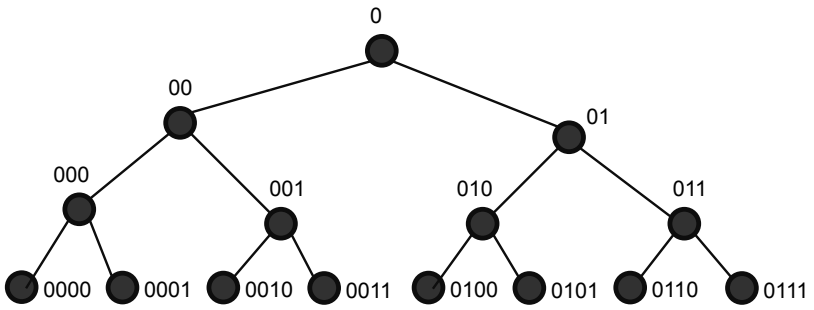


Fig. 3. Bit vector priorities in state space search

3.3. *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. It is often done in a work-pool based method where newly created tasks are queued in a work-pool for scheduling. Such work-pool based load balancing strategies have been studied extensively in the literature [9, 26]. Given the wide variety of load balancing algorithms, and the fact that each of them may work well for certain scenarios, users must be given the freedom to write strategies best suited to their applications, as well as the convenience of a set of commonly applicable schemes. ParSSSE is designed in a modular fashion so as to allow a clean separation of the load balancing procedure from the user code. The modular design makes it easy to

plug in an existing load balancing algorithm or develop a new one using a generic load balancing API.

We consider the design of the general and extensible load balancing framework next. ParSSSE allows fully distributed load balancing strategies to be plugged in at run-time. The load balancing module defines a core *task dispersal function*, which places a newly created task into a work-pool, either locally or on some other processor, depending on the load balance situation. When a new task is created by the program on a processor, the dispersal function may choose to place it into a pool of movable work. The work pool gradually shrinks as tasks are picked up and executed, but in most cases there is a sufficient amount of movable work which requires load balancing. As load conditions change, the load balancers may move the work in the pool across processors. Therefore, a new task may travel multiple hops among processors before it is executed. For reasons of efficiency, the number of hops may be limited by a chosen upper bound. This allows work to be assigned to the most lightly loaded processor, while ensuring that the overhead of doing so remains bounded.

Currently two load balancing strategies have been implemented in ParSSSE. The first one is a *randomized* load balancing strategy which assigns a newly generated task on a randomly selected processor. It is simple and has little overhead in making a load balancing decision. Given a sufficient large number of tasks, randomized placement tends to achieve a uniform distribution of tasks across processors. However, *randomized* strategy tends to incur higher communication overhead in the application, since a new task tends to be assigned to a remote processor which is far away from its original processor that it is likely to communicate with. Since the probability that each task is sent to a remote processor is $1 - \frac{1}{p}$ (where p is the total number of cores), when p is large, the probability approaches 1.

Randomized work stealing is a popular dynamic load balancing scheme, one whose asymptotic optimality has been established in the context of divide-and-conquer applications. It has been employed with some success in the Cilk [1] runtime system. Although originally intended for shared memory systems, randomized workstealing has recently been shown to improve performance on up to 8192 processors of a distributed memory system [4]. In distributed work stealing, a newly generated task is placed in the generating processor's *local* task queue. An idle processor (also called a *thief*, i.e. one with no tasks remaining in the local queue) randomly selects a *victim* processor to steal work from. If the victim's processor is not empty, it responds to the steal request by sending a set of tasks to the thief. Otherwise the victim sends the thief a negative acknowledgement, causing the thief to look for a different victim. Our implementation of the work stealing scheme extends this basic protocol by taking the priority of tasks into account. In particular, victims respond to thieves with tasks of higher priority, so that they may be executed as soon as possible on the thief processors. One advantage of work stealing is that it only moves tasks when processors are idle, which reduces the communication costs caused by moving the tasks away from its original processors. Section 5

compares the communication volume of randomized work stealing and randomized task placement.

Besides the two load balancing strategies above, new strategies can be easily implemented in ParSSSEby overriding the task dispersal function mentioned previously.

3.4. *Support for Branch-and-Bound Search*

The branch-and-bound technique is a very general tree-based search technique. In addition to adaptive grain size control, task prioritization and dynamic load balancing, a branch-and-bound procedure requires the timely pruning of unfit nodes, i.e. those nodes that have worse heuristic values than the current bound are not expanded. This reduces the amount of time spent in processing nodes that cannot possibly yield optimal solutions.

We follow an aggressive pruning strategy for branch-and-bound searches in ParSSSE. Nodes are pruned when they meet one of the following two requirements. First, a newly generated parallel or sequential node (task) is pruned when its lower bound is worse than the best solution. In this way, sub-optimal nodes do not consume much memory. Second, when a solution is found, its cost is compared with the cost of the best solution on its local processor. If the new solution is better, the cost is broadcast to all processors.

Finally, in order to reduce the communication latency of the bound information, two additional optimization techniques are applied in the ParSSSE. (1) The broadcast message containing the bound information is executed as soon as it arrives at a processor, even though that processor may have several tasks already waiting in the queue. (2) The broadcast operation is optimized using a spanning tree to reduce the communication overhead on the sending processor.

4. Application Programming Interface

In this section, we familiarize the reader with some of the specifics of the application interface presented by the search engine. An example application written using ParSSSE is also presented.

4.1. *Application Interface*

The code in Listing 1 shows the essential parts of the search engine framework. It also presents the handful of functions that must be implemented by the user in order to obtain a working parallel search program.

The *BTreeState* class must inherit from the *StateBase* class. It encapsulates all the data and operations 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 informs the framework of the user-defined static parallel search depth threshold,

Listing 1: Sample code written using ParSSSE

```
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);
```

which is discussed in Section 3.1. The *searchDepthLimit* function returns the initial maximum search depth if it is an Iterative Deepening A* (IDA*) search problem. The *lowerBound* function is only required in branch-and-bound problems in order to assign lower bound values to nodes. The *registerState* function returns a pointer to a newly generated state. Users do not have to explicitly allocate or manage the memory – this is done internally by the search engine. Finally, *process* is called after the child state has been set up and is ready to be searched.

Figure 4 illustrates the division of code and functionality between the user's program, ParSSSE and Charm++ runtime. From this figure, we can see that the user code is devoid of details about its parallel execution. Performance issues such as grain size control, prioritization and load balancing as described in Section 3 are handled by the search engine. Charm++ runtime provides the communication substrate and memory management functionality.

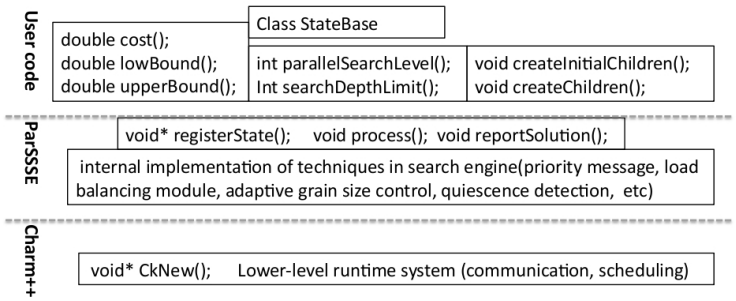


Fig. 4. ParSSSE Design and Implementation

4.2. *Performance Optimization*

In designing ParSSSE, our aim is to allow users to easily write state search programs using the framework to achieve the performance that matches specially tuned parallel programs. Therefore, two important concerns had to be addressed carefully in terms of productivity and performance. On the one hand, it is desirable that the framework provides a clean and simple interface that hides the implementation and optimization details from the users to enhance productivity, but on the other hand, extensive performance optimization tends to complicate the user interface. It is sometimes challenging to provide an abstract interface while minimizing the overheads.

As an example, through performance analysis, we found that a significant obstacle to achieve good performance is the copying of memory between buffers used by the framework and user space memory. Excessive memory allocation and memory copying incur significant overhead. In order to solve this problem, we designed a simple interface that exploits a LIFO stack data structure, which is managed by the framework. Meanwhile, interface functions are provided to the user code to access the data structure. Thus, when new nodes are generated, user code explicitly calls the interface functions to acquire memory from the stack. The memory region is manipulated by both user code and the framework without copying. Another optimization is that when the node on the top of the stack is processed, the memory region is recycled to a memory pool instead of being freed. Since creating and processing nodes are the dominant operations in state space search problems, this optimization saves a lot of time. Although optimizing the memory operations as described above exposes the LIFO stack data structure to the users, the user interface is carefully designed to be still simple with only two additional API functions introduced to manipulate the LIFO stack data structure.

5. Experimental Setup

In order to evaluate the performance of applications written with ParSSSE, we ported a number of state space search benchmark programs to the framework. Using these programs, we performed a number of strong scaling experiments on the *Intrepid* supercomputer, which is 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 PowerPC 450 cores running at 850 MHz. Each node has 2 GB of memory. Brief descriptions of the benchmarks follow.

Balanced Tree Search (BTS). This problem searches an abstract complete binary tree in which some leaf nodes are solutions. This program was used to illustrate anomalous speedups by Rao *et al.* [23], and therefore it serves as a good test of the task prioritization mechanisms in ParSSSE. Our test dataset is configured so that the first solution is located in the m-th leaf of the deepest level, where m is a pre-defined value.

N-Queens. N-Queens is a backtracking search problem in which N queens must be placed on an $N \times N$ chess board so that they do not attack each other. We search for all solutions to the N-Queens problem, for various values of N.

Unbalanced Tree Search (UTS). This is a parallel exploration of an unbalanced search tree[22]. This benchmark stresses the ability of ParSSSE to adaptively control grain size and perform load balancing proactively. Our experiments used two instances of the problem, namely *T1XXL* and *T3XXL*. The parameters of these instances (explained in [22]) are presented below:

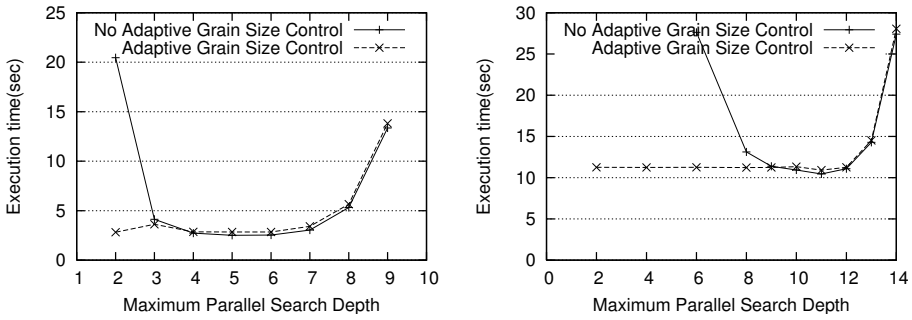
UTS Instance	t	a	d	b	r
T1XXL	1	3	15	4	19
T3XXL	0	2000	0.499995	2	318

Traveling Salesman Problem (TSP). Given a list of cities and their pairwise distances, the objective of this problem is to find a shortest possible tour that visits each city exactly once and returns to the starting city. The branch-and-bound technique is used to prune nodes aggressively and reduce the size of the search space. Although several sophisticated heuristics are available for the traveling salesman problem, in our experiments, we use a simple one based on the work of Little *et al.* [19], since our objective is to gain a sense of how well the framework performs on optimal solution searches.

6. Performance results

This section is presented in three parts, demonstrating the performance of ParSSSE in three different aspects, namely grain size control, task prioritization and dynamic load balancing.

Adaptive grain size control. In order to test the impact of adaptive grain size control on parallel performance, we ran the *18-Queens* and *T1XXL UTS* instances with different initial grain sizes and compared the execution times with static and adaptive grain size control schemes.



(a) 18-Queen (b) T1XXL

Fig. 5. Execution time on 1024 processors

Figure 5(a) shows this comparison for the *18 Queens* benchmark on 1024 processors. The x axis of this graph shows the maximum depth up to which parallel tasks are generated. Towards the left end, very few parallel tasks are created by the static grain size control mechanism, causing large amounts of load imbalance. The adaptive scheme does not suffer from this drawback, since tasks spawn new, parallel tasks after a certain amount of sequential work has been performed. Therefore, adaptive grain size control has significant benefits over the static scheme in this region. However, as the parallel task cutoff depth is increased, we see that the static scheme outperforms the dynamic one by about 10%. This is due to the overheads that the adaptive scheme incurs in instrumenting the sequential work. Notice that although the static scheme performs better in this region, the programmer must invest significant time and resources in determining the optimal cutoff depth. As maximum allowed depth is increased further (leading to finer grain sizes) we observe that both schemes suffer from the high overheads of parallelization. This suggests that it is a good strategy to first choose a small cutoff depth for parallel tasks, and then use adaptive grain size control to fine-tune performance.

Similar results were obtained for the T1XXL test (Figure 5(b)). In fact, given the large variations in tree depth, the advantage of using an adaptive grain size control mechanism is even more significant. For a different UTS instance T3XXL

that is a highly unbalanced search tree, different sets of static threshold from 10 to 200 are tried in static grain size control tests. The best result we got is around 600 seconds. However, the version using ParSSSE's adaptive control mechanism took only 7 seconds.

Task prioritization. We used the *BTS* benchmark to test how the prioritized execution helps reduce the number of speculative nodes processed by the search engine framework. We used trees of depth 30, thereby creating 2^{29} leaves. The search procedure looks for the first solution node at the deepest level. This experiment simulates heuristic-driven tree searches where solutions are more likely in the left portion of the tree.

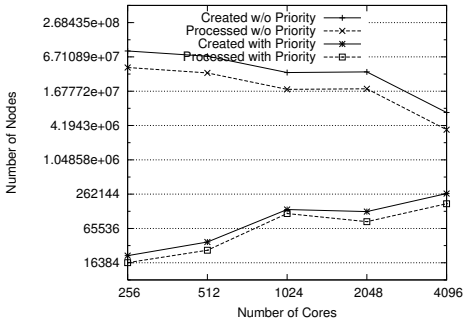


Fig. 6. The number of nodes created and processed in BTS with and without prioritized execution

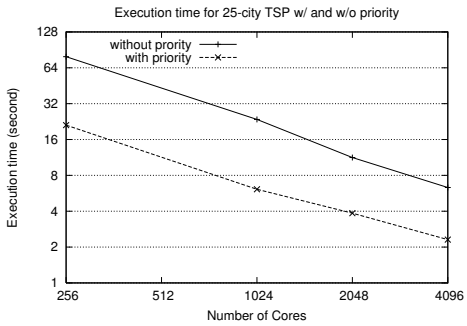


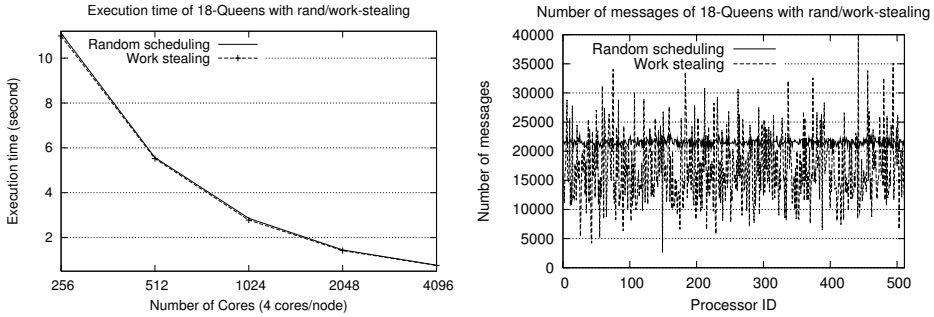
Fig. 7. The performance of a branch-and-bound TSP solver with and without prioritized execution

Figure 6 shows that prioritized execution helps to significantly reduce the number of nodes created and processed in balanced searches. Note that the difference between prioritized and non-prioritized execution becomes smaller as the number of processors increases. This is because the number of nodes per processor decreases as the number of processors increases, therefore, it takes less amount of time for the processors to find a solution, and so the saving of the prioritized execution is less significant.

In branch-and-bound problems like TSP, not only does prioritized execution help reduce speculative computation, it also helps find optimal solutions quicker. When the heuristic value of each node is equated with its priority, better nodes are processed first, thereby causing the global lower bound value to be reduced, so that more suboptimal nodes are pruned. This amplified effect of task prioritization can provide a significant boost to parallel performance. Figure 7 shows the execution time for a 25-city TSP problem with and without prioritized execution. It can be seen that prioritized execution improves performance by a factor of two across all processor counts.

Comparing different task scheduling strategies. As mentioned in section 3.3,

two task scheduling strategies are currently provided in ParSSSE. Whereas the strategies were compared qualitatively in that section, here we provide an empirical study of their differences. Figure 8(a) shows the execution times for the 18-Queens benchmark with random task scheduling and work stealing strategy respectively. Both strategies scale well from 512 to 4096 processors, although work stealing is slightly more efficient than the random strategy. Recall that work stealing leads to less communication overhead than the random strategy since tasks are moved only when thief processors become idle. This effect is illustrated in Figure 8(b) for the 18-Queens problem running on 512 processors. The total number of messages transferred across the network with the work stealing strategy is 30% lower than that with random balancing strategy.



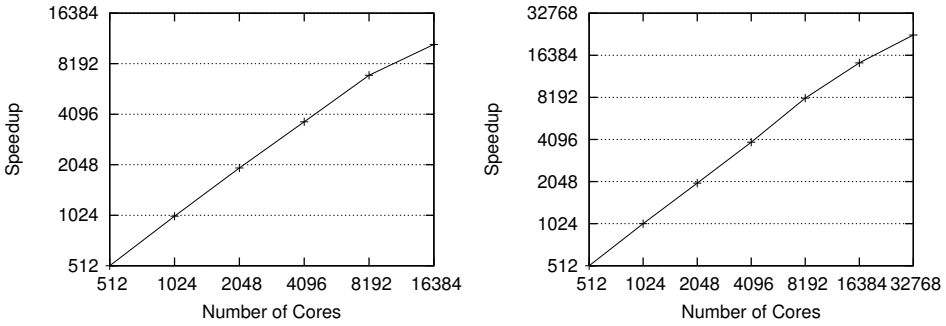
(a) (b)

Fig. 8. Comparing random and work stealing strategies. Subfigure (a) shows strong scaling profiles for random assignment and work stealing, whereas (b) compares the amount of communication generated by each strategy on 512 processors.

Strong scaling. To evaluate the overall scalability of programs written in ParSSSE, we performed strong scaling tests with the N-Queens and UTS benchmarks. Performance was measured on up to 16,384 processors for N-Queens and 32,768 processors for UTS. The results for *18-Queens* with an initial parallel cutoff depth of 5 are shown in Figure 9(a). On 8,192 processors the code achieved a parallel efficiency of 85.25% relative to execution time on 512 processors. For T1XXL instance shown in figure 9(b), ParSSSE achieved even better performance because of the higher ratio of computation to communication. Based on the performance on 512 processors, the efficiency on 8,192 processors was 98.75%, while on 16,384 processors it was still 88.28%. The code also scales well to 32,768 processors, with an efficiency of 69.78%.

7. 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 [7] and Nelson and Toptsis [21]



(a) 18-Queen (b) T1XXL

Fig. 9. Speedup from 512 processors to 32,768 processors

and, in the context of parallel logic programming, by Gupta *et al.* [8]. Particular paradigms in parallel combinatorial search have also been explored extensively – see [17, 16] for discussions of techniques and heuristics in branch-and-bound, [24] for parallel iterative deepening A*, [5] for game-tree search and [11] for parallel search of AND-OR trees. The importance of cutoff-based strategies in task parallel languages is highlighted in [20]. A discussion of the impacts of granularity on the performance of OR-parallel programs has been given by Furuichi *et al.* [6]. 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.* [13]. The use of work-stealing as a load balancing strategy was first described by Lin and Kumar [18] 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.

8. 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, and their corresponding techniques. We show how these techniques may be used to scale such applications to very large scale. We have incorporated these techniques into a general search engine framework ParSSSE, which 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 32,768 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] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 207–216, Santa Barbara, California, July 1995. MIT.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[3] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[4] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[5] Rainer Feldmann, Peter Mysliwiete, and Burkhard 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*, pages 94–103, New York, NY, USA, 1994. ACM.

[6] 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*, pages 50–59, 1990.

[7] Ananth Grama and Vipin Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, 1999.

[8] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

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

[10] Laxmikant V. Kale and Gengbin Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[11] L.V. Kalé. The REDUCE OR process model for parallel execution of logic programs. *Journal of Logic Programming*, 11(1):55–84, July 1991.

[12] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[13] L.V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.

[14] L.V. Kalé and V. Saletore. Parallel state-space search for a first solution with consistent linear speedups. *Internaltional Journal of Parallel Programming*, 19(4):251–293, 1990.

[15] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *J. ACM*, 40(3):765–789, 1993.

[16] Ten-Hwang Lai and Sartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *Commun. ACM*, 27:594–602, June 1984.

[17] Guo-Jie Li and Benjamin W. Wah. Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Transactions on Computing*, 35(6):568–573, 1986.

[18] Yow-Jian Lin and Vipin Kumar. And-parallel execution of logic programs on a shared-memory multiprocessor. *J. Log. Program.*, 10(1/2/3&4):155–178, 1991.

[19] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989, 1963.

[20] Hans-Wolfgang Loidl and Kevin Hammond. On the granularity of divide-and-conquer parallelism. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1995.

[21] Peter C. Nelson and Anestis A. Toptsis. Unidirectional and bidirectional search algorithms. *IEEE Software*, 9:77–83, March 1992.

[22] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *Lecture Notes in Computer Sciences*, volume 4382, pages 235–250. Springer-Verlag, 2007.

[23] V. Nageshwara Rao and Vipin Kumar. Superlinear speedup in parallel state-space search. In *Proceedings of the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 161–174, London, UK, 1988. Springer-Verlag.

[24] V. Nageshwara Rao, Vipin Kumar, and K. Ramesh. A parallel implementation of Iterative-Deepening-A*. In *AAAI*, pages 178–182, 1987.

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

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

[27] A. Sinha and L.V. Kalé. A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA., April 1993.