Using Load Balancing to Scalably Parallelize Sampling-Based Motion Planning Algorithms

Adam Fidel[†], Sam Ade Jacobs[*], Shishir Sharma[‡], Nancy M. Amato[†], Lawrence Rauchwerger[†]

***Abstract*—Motion planning, which is the problem of computing feasible paths in an environment for a movable object, has applications in many domains ranging from robotics, to intelligent CAD, to protein folding. The best methods for solving this PSPACE-hard problem are so-called sampling-based planners. Recent work introduced uniform spatial subdivision techniques for parallelizing sampling-based motion planning algorithms that scaled well. However, such methods are prone to load imbalance, as planning time depends on region characteristics and, for most problems, the heterogeneity of the subproblems increases as the number of processors increases. In this work, we introduce two techniques to address load imbalance in the parallelization of sampling-based motion planning algorithms: an adaptive work stealing approach and bulk-synchronous redistribution. We show that applying these techniques to representatives of the two major classes of parallel sampling-based motion planning algorithms, probabilistic roadmaps and rapidly-exploring random trees, results in a more scalable and load-balanced computation on more than 3,000 cores.**

I. Introduction

While motion planning has its roots in robotics, it now finds applications in other areas of scientific computing including protein folding [30], drug design [29] and virtual prototyping and computer-aided design [8].

Due to the infeasibility of exact motion planning, sampling-based methods are now the state-of-the-art for solving motion planning problems. Sampling-based motion planning is essentially a large-scale graph problem that first constructs a graph and subsequently solves queries using graph traversals. Sampling-based motion planning algorithms have been highly successful at solving previously unsolved problems [10], and much research has focused on developing more sophisticated variants of them. Sampling-based approaches are efficient and can be applied to problems with many degrees of freedom (e.g., robotic manipulators with many links or proteins with many amino acids). While not guaranteed to find a solution, they are known to be probabilistically complete, meaning that the probability of finding a solution, given one exists, increases with the number of samples generated [23].

Even so, substantial resources in time and hardware are still required to solve complex applications. For example, modeling the motion of a small protein using sequential sampling-based motion planning techniques can take days on a typical desktop machine [31]. Thus, it is practically infeasible to study larger proteins or to significantly increase the detail and accuracy at which their motions are modeled. Hence, there is a need for more efficient methods and parallel processing is a natural option to explore.

Previous work [19], [20] proposed methods based on uniform workspace subdivision for parallelizing representatives of the two major classes of sampling-based motion planning algorithms. By subdividing the space and restricting the locality of graph connection attempts, inter-processor communication associated with nearest neighbor searches – a well-known bottleneck in parallelizing sampling-based motion planning algorithms [2], [11], [14], [27] – can be substantially reduced. This approach achieves better and more scalable performance on different parallel machines than previous methods. Fundamentally, uniform spatial subdivision methods are limited in the types of motion planning problems they can solve efficiently. In particular, for most non-trivial environments, as the problem is subdivided, the variance in the amount of work performed by the subdivisions will increase. Because of the difference in complexity of the subdivided regions, there will be a corresponding increase in load imbalance.

In this work, we apply and analyze two techniques to address the problem of load imbalance for parallel sampling-based motion planning algorithms. The first is an adaptive work stealing approach that permanently migrates both the region and the work related to it to improve the load balance. The second approach is a bulk-synchronous redistribution technique that redistributes regions among processors to have a more balanced distribution of data. We propose a method based on the complexity of a region to approximate the amount of work per region and use it to attempt to balance work across processors, while also preserving the

[†]*Parasol Lab*, Dept. of Computer Science and Engineering, Texas A&M University, 3112 TAMU, College Station, TX, USA. {fidel,amato,rwerger}@cse.tamu.edu

[*]*ABB Corporate Research*, Raleigh, NC, USA. sam.jacobs@us.abb.com. This work was completed while a member at *Parasol Lab*.

[‡]*Microsoft Corp.*, Redmond, WA, USA. shishir.sharma@microsoft.com. This work was completed while a member at *Parasol Lab*.

This research supported in part by NSF awards CNS-0551685, CCF-0833199, CCF-0830753, IIS-0916053, IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, B575363, by Samsung, Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

spatial geometry of the subdivision.

Contribution. Key contributions of this paper include:

- A study of the limitations of uniform spatial subdivision for parallel sampling-based motion planning;
- Application of load balancing techniques based on graph redistribution and work stealing to combat load balancing issues that arise at scale;
- A theoretical analysis of load imbalance for a model environment and bounds on the improvement that any load balancing technique can achieve for parallel sampling-based motion planning;
- Demonstration of the performance benefit of load balancing strategies across various workloads on several parallel architectures ranging from Linux clusters to a Cray XE6 petascale machine and an experimental evaluation on more than 3,000 cores.

II. Preliminaries

This section reviews basic load balancing techniques and provides an overview of previous work in parallel sampling-based motion planning.

A. Load Balancing Techniques

Work stealing [3], [4] is an important technique used to balance an imbalanced computation. In this method the computation is logically divided into a collection of tasks. When a processing element runs out of its local tasks, it attempts to steal tasks from potential victims. This strategy is well suited for shared-memory systems. In distributed-memory systems, there are two variations on the way data can be made available to the thief: replication and ownership transfer. In the case of replication, some sort of software coherence mechanism may be required to deal with the multiple copies of data. In the case of ownership transfer, the overheads associated with transferring ownership to the thief processor need to be managed. In this work, we have a model in which transfer of ownership is considered.

Repartitioning of the data is another strategy to address load imbalance. It is well known that data distribution is fundamental to achieving acceptable levels of load balance. There exists a large body of literature regarding partitioning of distributed data structures [12], [22]. We focus on computing, and enforcing through data migration, high quality partitions of the problem across processing elements.

In general, the type of load balancing technique applied to an imbalanced computation depends on the nature of the computation itself. Repartitioning is well suited for applications in which a good estimate of the computation associated with the data can be easily computed and the total amount and structure of the computation is known *a priori*. In contrast, work stealing is best suited for dynamic applications in which either the execution of the algorithm defines more computation as the algorithm progresses, or the work associated with the task cannot be easily estimated.

B. Parallel Sampling-Based Motion Planning

The motion planning problem is to find a valid path (e.g., one that is collision-free and satisfies any joint limit and/or loop closure constraints) for a movable object starting from a specified start configuration to a goal configuration in an environment [10]. A single configuration is specified in terms of the movable object's d independent parameters or degrees of freedom (DOF). The set of all possible configurations (both feasible and infeasible) defines a configuration space ($\mathbb{C}_{space}$). $\mathbb{C}_{space}$ is partitioned into two sets: $\mathbb{C}_{free}$ (the set of all feasible configurations) and $\mathbb{C}_{obstacle}$ (the set of all infeasible configurations). Motion planning then becomes the problem of finding a continuous sequence of points in $\mathbb{C}_{free}$ connecting the start and goal configuration.

A complete solution of the motion planning problem is considered computationally intractable and has been shown to be PSPACE-hard with the best known upper bound being doubly exponential in the movable object's DOFs [28]. As an alternative, randomized and approximate solutions have been shown to be efficient and practical. Sampling-based methods [10] are the state-of-the-art approach to solving motion planning problems in practice. Sampling-based methods are known to be probabilistically complete, i.e., the probability of finding a solution given one exists increases with the number of samples generated. Sampling-based methods are broadly classified into two main classes: roadmap or graph-based methods such as the Probabilistic Roadmap Method (PRM) [23] and tree-based methods such as the Rapidly-exploring Random Tree (RRT) [24]. In the following, we describe previous approaches to parallelize methods from both classes.

1) Parallelizing PRM with Uniform Subdivision: The Probabilistic Roadmap Method (PRM) [23] constructs a graph $G = (V, E)$, called a roadmap, to capture the connectivity of $\mathbb{C}_{free}$. A node in G represents a valid configuration and an edge represents a valid trajectory (path) between configurations. Nodes are generated using some sampling strategy and connections are attempted between a node and its k-nearest neighbors as computed using some distance metric. Once the roadmap is constructed, query processing is done by connecting the start and goal configurations to the roadmap and extracting a path through the roadmap that connects them.

A uniform $\mathbb{C}_{space}$ subdivision method for parallelizing PRM was presented in [19] and is shown in Algorithm 1. In line 2 of Algorithm 1, the $\mathbb{C}_{space}$ representing the movable object is subdivided into regions. For example, in three-dimensional environments, the planning space may be subdivided into regions using the $\mathbb{C}_{space}$ positional degrees of freedom, i.e., the x, y and z dimensions. A simple illustration of a 2D environment subdivided into four regions is shown in Figure 1(a). The subdivision is represented by a *region graph*, whose vertices represent regions and whose

edges encode the adjacency information between regions. Figure 1(b) shows the region graph corresponding to the subdivision shown in Figure 1(a).

Algorithm 1 Uniform Subdivision

Input: An environment env, the number of nodes N, the number of processes p, the number of regions N_r
Output: A roadmap graph G

1: Let region graph $R(V, E) = \emptyset$.
2: Let R_d = Subdivide env into N_r regions.
3: Add a vertex for each region r of R_d to R.
4: **for all** neighboring regions $(r_1, r_2) \epsilon R_d$ **par do**
5: Add the edge (r_1, r_2) to R.
6: **end for**
7: **for all** regions $r_i \in V$ **par do**
8: $G \leftarrow$ Independently construct regional roadmap using sequential PRM
9: **end for**
10: **for all** neighboring regions $(r_i, r_j) \in E$ **par do**
11: $G \leftarrow$ Connect regional roadmap of regions r_i and r_j
12: **end for**
13: **return** G

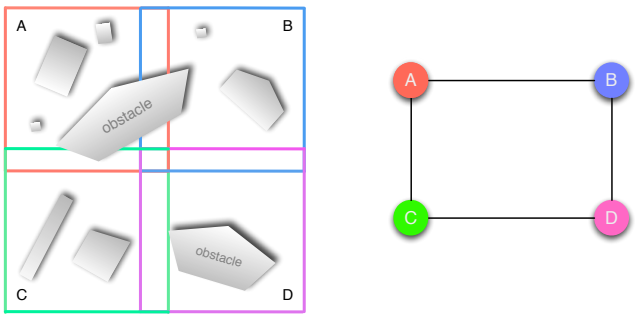

Figure 1. A 2D environment subdivided into 4 regions and the corresponding region graph.

In line 8 of Algorithm 1, roadmaps are constructed in parallel in each region. This is done by invoking the (sequential) PRM planner [23] in each region. Lastly, in lines 10 – 12, the regional roadmaps are connected to form a roadmap of the entire $\mathbb{C}_{free}$. The region graph facilitates the process for connecting regional roadmaps by identifying adjacent regions between which connections are attempted. Some user-defined overlap is allowed between regions to allow sampling in the portion of space at the boundaries that may facilitate connections between regional roadmaps.

2) Parallelizing RRT with Uniform Radial Subdivision: The Rapidly-exploring Random Tree (RRT) method is another sampling-based motion planning approach particularly well suited for non-holonomic and kinodynamic motion planning problems [24]. The basic sequential RRT grows a tree rooted at the start configuration that expands outward into unexplored areas of $\mathbb{C}_{space}$. To build a tree, RRT first generates a uniform random sample q_{rand}, and identifies the closest node q_{near} in the tree to q_{rand}, and then q_{near} is "extended" toward q_{rand} for a stepsize of at most Δq. If successful, q_{new} is added to the tree as a node and the pair (q_{near}, q_{new}) is added as an edge. To solve a particular motion planning query, RRT repeats this process until the goal configuration can be connected to the tree.

Uniform radial subdivision [20] is particularly suited for parallelizing RRTs. In their method, $\mathbb{C}_{space}$ is subdivided into conical regions and subtrees are built in each region using the sequential RRT planners. Similar to uniform subdivision described earlier, regional subtrees are later connected to subtrees in neighboring regions. A 2D illustration of radial subdivision of $\mathbb{C}_{space}$ is shown in Figure 2.

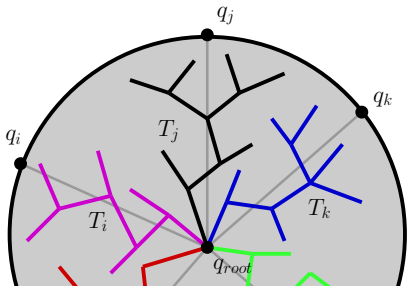

Figure 2. Example of uniform radial subdivision for a 2D $\mathbb{C}_{space}$. Each process concurrently builds a branch (using sequential RRT) rooted at q_{root} and biased toward a target (e.g., q_k for the blue process).

The uniform radial subdivision algorithm is shown in Algorithm 2. Lines 1 and 2 describe the region construction phase. First, a hypersphere is created in d-dimensional $\mathbb{C}_{space}$ S^d centered at $q_{root} \in R^d$ with radius r. Next, N_r points $q_i \in R^d$ are sampled on the surface of S^d. Each point defines a conical region centered around the ray $\overrightarrow{q_{root}q_i}$. A region graph $G(V, E)$ is then constructed. Each vertex $v_i \in V$ represents a region defined by q_i and an edge (v_i, v_j) is added if q_j is in the k closest neighbors of q_i.

After region graph construction, independently (in parallel) a sequential RRT is used in each region. The subtree growth in each region is biased toward the region candidate defined by the random ray $\overrightarrow{q_{root}q_i}$. Some overlap between regions is allowed so branches can explore part of the space in adjacent regions. Lastly, using the adjacency information provided by the region graph, connection attempts are made between each region branch and the branches in adjacent regions. If any edge connection creates a cycle, the tree is pruned so as to remove the cycle.

III. Load Balancing For Parallel Motion Planning

Uniform (radial) subdivision is limited in the types of motion planning environments it can handle. It performs well in uniform and homogeneous environments, but not so

Algorithm 2 Uniform Radial Subdivision

Input: An environment env, a root q_{root}, the number of nodes N, the number of processes p, the number of regions N_r, a region radius r, the number of adjacent regions k

Output: A tree T containing N nodes rooted at q_{root}

1: Construct a sphere S^d rooted at q_{root} with radius r
2: $Q_{N_r} \leftarrow$ sample N_r random points on the surface of S^d
3: Initialize region graph $G(V, E)$ with $V \leftarrow Q_{N_r}$ and $E \leftarrow \emptyset$
4: **for all** points $q_i \in Q_{N_r}$ **par do**
5: $neighbors \leftarrow$ FindNeighbors(G, q_i, k)
6: **for all** $n \in neighbors$ **do**
7: G.AddEdge(q_i, n)
8: **end for**
9: **end for**
10: **for all** regions $v_i \in V$ **par do**
11: $T \leftarrow$ Independently construct regional subtree using sequential RRT
12: **end for**
13: **for all** neighboring regions $(v_i, v_j) \in E$ **par do**
14: $T \leftarrow$ Connect regional subtrees of regions v_i and v_j
15: **if** Cycle(T) **then**
16: Prune(T)
17: **end if**
18: **end for**
19: **return** T

well in non-uniform and heterogeneous environments. For example, a house or factory floor is typically composed of logically separate parts; open or free space, cluttered space, doorways, etc. Uniform subdivision in this scenario is prone to load imbalance. As an illustration, consider the uniform subdivision of the planning space in Figure 3(a); if different processors are assigned to each region, processors assigned to region R_0 are relatively overloaded. This irregularity in planning space leads to workload imbalance, which will have an overall negative affect on scalability.

Figure 3 illustrates the roadmap distribution for an environment that suffers from a high degree of load imbalance using uniform spatial subdivision. Shown is a sample run with four processors where the color of a node represents a single processor. Figure 3(b) shows that the majority of the roadmap nodes are only present on two processors, and the remaining two processors have only a small number of vertices. In contrast, Figure 3(c) shows an even distribution of roadmap nodes after applying load balancing techniques.

One important consideration is the granularity in which the problem is partitioned, as the size of the biggest quanta of work establishes a lower bound by which the problem can be balanced using any load balancing strategy. In addition, a more refined problem provides more opportunity to distribute work amongst processing elements. For parallel

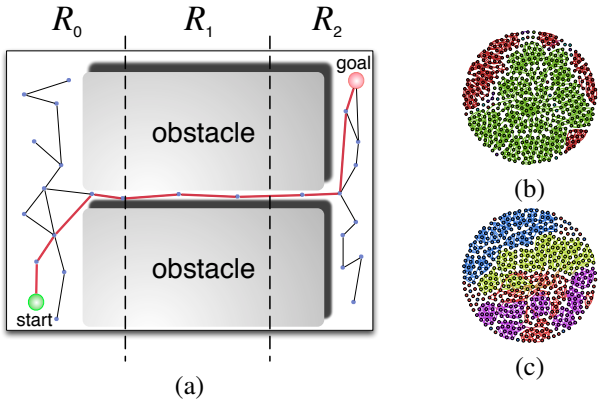

Figure 3. An (a) imbalanced environment and roadmap graph node distribution (b) before and (c) after rebalancing for parallel PRM.

motion planning, regions represent the quanta of work and thus for the presented load balancing strategies, we consider an over-partitioned region graph.

In general, a load balancing strategy using repartitioning requires a reasonable estimate for the amount of effort that is required to compute a quanta of work. In section III-B, we will discuss weighting techniques for the two discussed parallel motion planning algorithms and the difficulty of estimating work for uniform radial subdivision RRT.

A. Work Stealing for Parallel Sampling-Based Planning

Work stealing [4] is a well-known load balancing technique that does not require an estimate of the cost of a quanta of work. In parallel sampling-based motion planning, work stealing can be applied to either the uniform subdivision PRM or uniform radial subdivision RRT algorithms.

Algorithm 3 Work stealing Parallel Sampling-Based Motion Planning

Input: Region graph, sequential planner, steal policy.

Output: Set of constructed RRT branches or PRM roadmaps

1: **while** Global termination not detected **do**
2: **for all** $p \in Processors$ **par do**
3: $Q \leftarrow$ { Regions of p }
4: **while** Q is not empty **do**
5: $R_{current} \leftarrow$ DEQUEUE(Q)
6: $T \leftarrow$ Independently construct regional subtree or roadmap using sequential planners using $R_{current}$
7: **end while**
8: $V \leftarrow$ choose victim based on steal-policy
9: Steal regions from V based on policy
10: **end for**
11: **end while**

Algorithm 3 shows a generic work stealing approach for parallel sampling-based motion planning, highlighting

opportunities to steal during either the construction of RRTs for uniform radial subdivision or a roadmap for PRM. The main computation in which the (sequential) planner is invoked is shown in Line 6. As each processor is assigned regions in which to explore, we model these regions in a local work queue. When this local queue is depleted, the processing element will issue steal requests to potential victims. On a victim processor, work is stolen from the back of its local work queue.

Victim selection is a particularly important decision, primarily because the cost of stealing from a processor on the same shared-memory node is generally less than the cost of stealing from a processor on another node. Additionally, for parallel motion planning, the choice of victims should also be related to the distribution of the region graph among the processors. In particular, since neighboring processing elements will communicate with each other to perform region connections after the RRT expansion or PRM construction phase, stealing from neighbors would benefit the region connection phase because connecting regions will be local to the same processing element.

We consider three work stealing strategies in the context of parallel motion planning. One strategy (RAND-K) is a randomized strategy in which a thief requests additional regions from k random processors, but not necessarily the same k processors for each request. For the purpose of our experimental evaluation, we have fixed k to be 8. Another strategy is DIFFUSIVE wherein processors are assumed to be arranged in a 2D mesh and underloaded processors will request neighboring processors for work. Finally, we consider the HYBRID strategy wherein we first execute DIFFUSIVE stealing and in the event that no request could be serviced, requests are sent to random processors. In the experimental results section, we compare and contrast these strategies.

B. Repartitioning for Parallel Sampling-Based Planning

In Algorithm 4, we show how to use repartitioning to influence load balancing in parallel PRM and RRT. The main imbalanced computation, construction of independent roadmaps or RRTs for a given region, is performed only after attempting to redistribute the region graph based on the weight for each region. This will balance this phase of computation according to a suitable *estimate* of work to compute a region. Note that lines 5-7 in the algorithm can be used in lines 7-9 in the original parallel PRM algorithm (Algorithm 1), and we simply redistribute the regions before constructing individual roadmaps. Similarly, lines 5-7 in the algorithm can be plugged into lines 10-12 in the uniform radial subdivision RRT algorithm (Algorithm 2), provided a suitable weight for a region can be computed.

The effectiveness of repartitioning is highly dependent on the ability to estimate the load of an RRT or PRM region.

Algorithm 4 Repartitioning for Parallel Sampling-Based Motion Planning

Input: Regional graph, sequential planner
Output: Set of constructed RRT branches or PRM roadmaps

```
1: W ← ∅
2: for all regions v_i ∈ V par do
3:     W_i ← ComputeRegionWeight(v_i)
4: end for
5: GraphRepartition(R, W)
6: for all regions v_i ∈ V par do
7:     G ← Independently construct regional roadmap or
       RRT using sequential planners
8: end for
```

PRM. For uniform subdivision PRM, since we can easily and cheaply compute a cost metric of the amount of work to be done, this algorithm is a good candidate to use repartitioning. In parallelizing PRM, the two data structures of interest are the graph representation of the regions and the PRM roadmap. The regions represent a spatial subdivision of the environment in which configurations will be sampled. Connections are attempted between configurations through the use of local planning methods. It is well known in motion planning that the cost of connecting samples in $\mathbb{C}_{space}$ is highly representative of the amount of time the overall algorithm will take in generating a solution [31]. This in fact is the most time consuming phase of the entire computation. As regions that have a high number of samples will generally incur a large number of local planning calls, a good metric for approximating the amount of work that a region will generate is the number of samples in the roadmap that lie within that region.

Using this information, we can determine that load imbalance in terms of regions corresponds to the number of roadmap samples of the region, and this metric can be used to weight regions. A high quality partition of the region graph will attempt to balance the regions based on this metric. However, as regions are also spatial entities, the spatial geometry of regions should also be preserved in an ideal partition. By partitioning the region graph using these approximations of the amount of work that a region will perform, the algorithm will see a higher level of load balance for subsequent phases of computation.

RRT. Unfortunately, in uniform radial subdivision RRT, the amount of work a region will perform is difficult to estimate beforehand. In our experiments, we show an estimate of work for an RRT branch that uses k random rays originating from the origin of the region, and computes the minumum distance to an obstacle in the direction of these rays. Intuitively, this should give a reasonable approximation of the amount of *reachable* free space in that region; however, we show that this metric is a poor indicator of

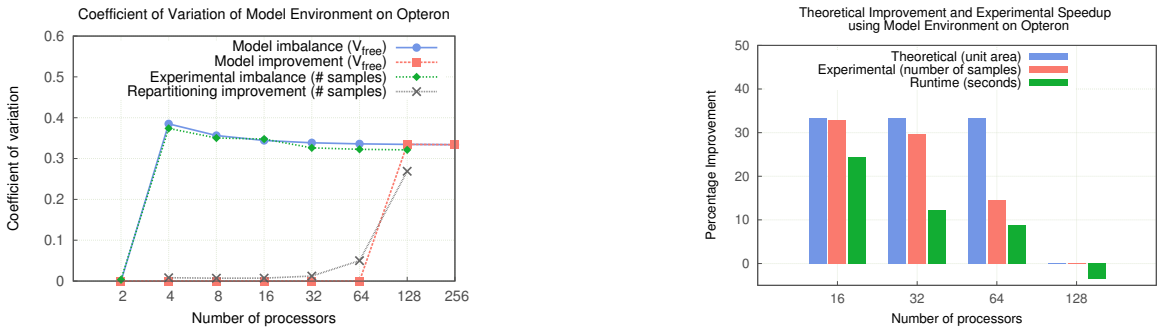

Figure 4. Experimental validation of (a) measure of load imbalance and (b) potential improvement in model environment.

work for a given region unless a large number of rays is utilized, making this an expensive operation to calculate.

IV. Experimental Evaluation

We implement and evaluate standard load balancing techniques for parallel motion planning and show that with an appropriate estimate for the amount of work in a region, geometric repartitioning outperforms work stealing. In the dynamic case where load is unknown *a priori*, repartitioning will be at a disadvantage and can potentially be worse than performing no load balancing at all.

Experimental studies were conducted on two massively parallel machines: a 153,216 core Cray XE6 (HOPPER) and a 2,400 core Opteron cluster (OPTERON-CLUSTER). Unless otherwise noted, all experiments show strong scaling in which the total number of regions is kept constant as the number of processing elements increases.

A. Implementation in STAPL

The parallel motion planning algorithms and load balancing techniques described have been implemented using STAPL [5], a generic, scalable framework for parallel C++ code development. At STAPL's core is a library of C++ components implementing parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`) that have interfaces similar to the (sequential) C++ standard template library (STL) [25]. Parallel algorithms are expressed as arbitrary task dependence graphs in STAPL.

Load imbalance in parallel computations is dealt with in various ways in STAPL. For repartitioning, this is realized through redistribution of the two `pGraphs` [16] (i.e., the region graph and the roadmap or RRT graph) in the parallel motion planning algorithms. Alternatively, load balancing can be addressed by using a custom work stealing scheduler for parallel motion planning algorithms.

B. Model PRM Environment

Consider a 2D environment with a single square obstacle that lies equidistant from the bounding box. It is possible to compute the volume of the free space (V_{free}) by using the total volume of the region and the volume of the obstacle within the region. With an estimate of the free space in the environment, we can say that the total load that the region will experience is proportional to V_{free}.

One measure of imbalance among processors is the *coefficient of variation*, defined to be the ratio of the standard deviation σ and mean μ load. A naïve mapping of regions to processors would perform a 1D partitioning of the region mesh and assign a balanced number of region columns to processors. This naïve region mapping will have a high coefficient of variation for the model environment. We find an estimate of the most balanced partitioning of the region graph statically ignoring edge-cuts using a greedy global partitioning algorithm, as the exact problem is NP-complete.

Figure 4(a) shows the model's prediction of the imbalance in terms of the coefficient of variation of samples (lower is better) with the naïve partitioning strategy and the best load balance possible. In addition, we plot the measure of load imbalance experienced during a trial run of the algorithm using repartitioning and show that we closely track the model. As shown, the best possible distribution of regions to processors for higher core counts shows less benefit, as each processor has an increasingly smaller granularity of work as the number of processors increases.

Figure 4(b) studies various metrics according to the model and an experimental evaluation. We study the potential improvement according to the model (theoretical), which measures the total reduction in V_{free} for the processor with the highest amount of V_{free}. Next, we plot the reduction in the number of roadmap nodes (experimental) on the highest loaded processor. Finally, we show the overall improvement in execution time (runtime) for the load-balanced phase using repartitioning. In general, we track the model's theoretical estimate of the best load distribution in terms of roadmap nodes, which in turn tracks the improvement in execution time. The discrepancies between the best distribution of V_{free} and the roadmap node distribution can be explained by both the probabilistic nature of the computation and by

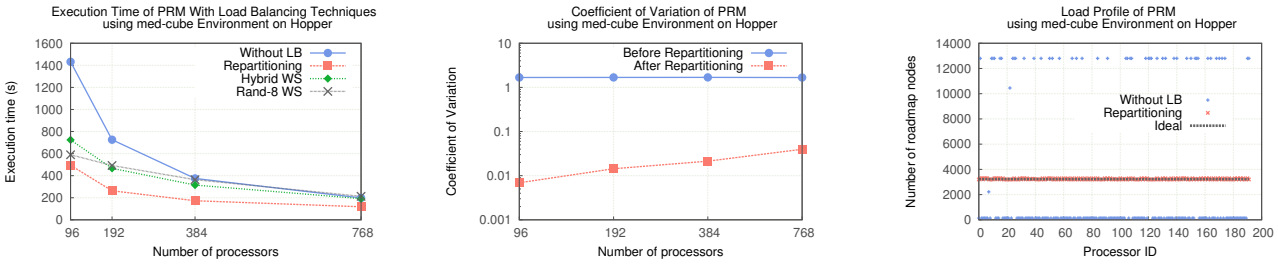

Figure 5. Evaluation of (a) execution time and (b) coefficient of variation and (c) load distribution for PRM on HOPPER using med-cube.

the geometric restrictions enforced by the repartitioning. The gap between the improvement in roadmap distribution and total time reduction is a result of the number of roadmap nodes per region being an imperfect indicator of the total amount of work generated by that region. At 128 cores, there is no better distribution of load possible, so the experimental result only shows the overhead of attempting to repartition.

C. Experimental Results

1) Repartitioning: **PRM**. The environments considered in this section are variants of a 3D narrow passage with a rigid-body robot, similar to the theoretical environment that consists of a single cubic obstacle in which roughly 24% (med-cube), 6% (small-cube) and 0% (free) of the environment is blocked. In all environments, we subdivide the problem into 250,000 regions total. Figure 5(a) shows raw execution time for computing the final roadmap on the HOPPER platform for this strong scaling PRM experiment in the med-cube environment. We can see that using repartitioning, we are able to achieve a 2.9x improvement over the baseline on 96 cores and a 1.68x improvement on 768 cores. Because of the strong scaling nature of our experiment, there are significantly fewer regions per processor at 768 cores, which allows for less opportunity for moving load across processors. From Figure 5(b), we can see that although the coefficient of variation is substantially lower for all processor counts after repartitioning, the difference is not as much for higher processors counts simply because of a reduced opportunity to rebalance. Figure 5(c) shows the distribution of load across processors on a 192-core run on HOPPER. We see that without load balancing, there is a wide spread in work and after applying repartitioning, a distribution closer to the ideal is achieved. Figure 6 shows the trend shown in the previous analysis holds for higher processor counts on HOPPER.

For the same experiment, we show the breakdown of the various phases of parallel PRM in Figure 7(a). As suspected, the portion of the computation connecting roadmap nodes in a region (line 8 in Algorithm 1) dominates most of the computation at 90% of the total execution time. After load balancing for both methods, the total time decreases,

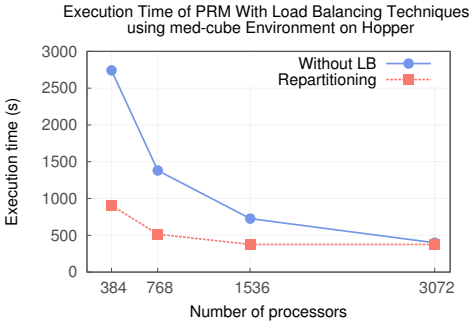

Figure 6. Evaluation of computing roadmap in the med-cube environment for a rigid body robot on Hopper

mainly because of the decrease in node connection time. For repartitioning, there is an increase in region connection time (line 11 in Algorithm 1), which can be partially attributed to an increase in remote accesses in the region connection phase, as shown in Figure 7(b). This is due to an increase in edge cuts, which was induced by repartitioning.

Figure 8 demonstrates load balancing techniques on multiple environments that display different levels of imbalance on OPTERON-CLUSTER. In Figure 8(a) and (b), we see up to a 3.4x speedup over the baseline using repartitioning in the med-cube environment, but only a 1.2x speedup in the small-cube environment. This shows that even on workloads that are not imbalanced to such a high degree, load balancing can still provide a large benefit. In addition, we find that in the free environment which exhibits no imbalance, all load balancing techniques do not show any significant overhead over the non-load balanced variant, as shown in Figure 8(c).

RRT. We also evaluated load balancing techniques on the uniform radial RRT parallel motion planning algorithm. As discussed in Section III-B, it is difficult to estimate the amount of work that a radial branch will compute due to the probabilistic and dynamic nature of the algorithm. Thus, computing an effective partition for load balancing is difficult and may prove to be inaccurate.

In Figure 10(b), we computed an estimate of work for an RRT branch by using the k random rays metric discussed

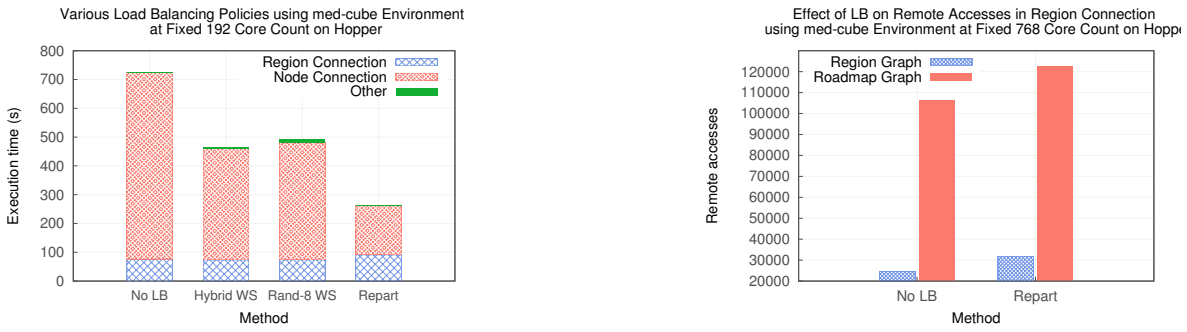

Figure 7. Breakdown of (a) the various phases of PRM (b) and the effect of load balancing on remote accesses.

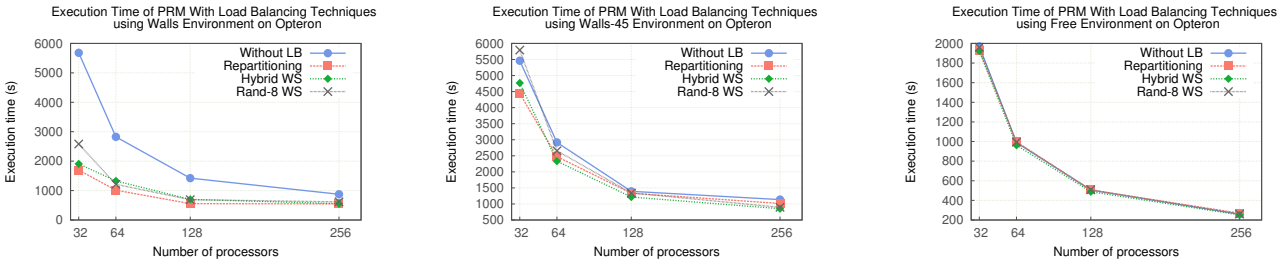

Figure 8. Execution time for PRM with various load balancing strategies in (a) med-cube (b) small-cube (c) and free environment.

in Section III-B. This metric should intuitively estimate the number of local planning calls in the RRT construction, yet it acts as a poor estimate for the amount of work needed to compute the RRT branch and in most cases, we see a slowdown when using this weight. Indeed, any metric to estimate the amount of work in this random and dynamic algorithm would likely be imprecise and it is for this reason work stealing strategies are better suited for uniform subdivision radial RRT.

2) Work Stealing: **PRM**. In the PRM environments, we see that HYBRID work stealing outperforms the RAND-K strategy, mainly due to the non-deterministic behavior of RAND-K and the low probability of finding work using a random strategy. Figure 9 provides a breakdown for HYBRID work stealing illustrating the number of tasks that were executed locally and the number of stolen tasks for each processor. In Figure 9(a), we see that a substantial number of underloaded processors find work to be stolen and execute a large amount of stolen tasks. In contrast, at higher processor counts, such as those shown in Figure 9(b), it becomes difficult for underloaded processors to find work to be stolen, as the work per processor decreases and the pool of potential processors from which to request increases. Experimentally, few processors are able to find work once they have exhausted their local regions. Moreover, the amount of work available for stealing also decreases. Both of these behaviors are expected for strong scaling experiments.

In general, work stealing strategies for PRM perform better than the non-load-balanced run, but not as well as repartitioning. In the breakdowns shown in Figure 7, we can see that the node connection phase does not improve to the extent of repartitioning, due to the random and non-exact nature of work stealing and various overheads involved. However, region connection was not affected to the degree seen with repartitioning because the method ultimately did not move a large number of regions and thus the number of edge cuts were not affected as severely.

RRT. We evaluated work stealing on the radial RRT parallel motion planning algorithm. Figure 10 shows the total execution time for computing the final RRT for a rigid body robot in two cluttered environments and one free environment on OPTERON-CLUSTER. We varied the amount of free space in each environment such that the first environment (mixed) is 60% blocked, the second environment (mixed-30) is 30% blocked and the third environment (free) is completely free of obstacles (0% blocked). Using the DIFFUSIVE work stealing strategy allowed the algorithm to achieve a speedup of 2.0x on 32 cores and 1.55x at 256 cores in the mixed environment. A similar pattern of decreasing marginal benefit of work stealing from uniform subdivision is exhibited in this experiment. As with uniform subdivision, the stealable work per processor decreases with the number of processors, while the number of potential victims from which to steal also increases.

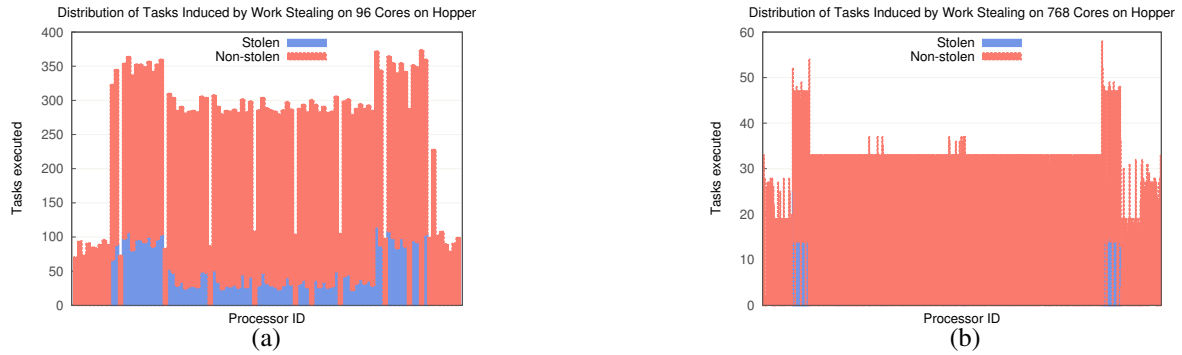

Figure 9. Breakdown of the amount of tasks stolen vs. executed locally for PRM on (a) 96 and (b) 768 cores on HOPPER.

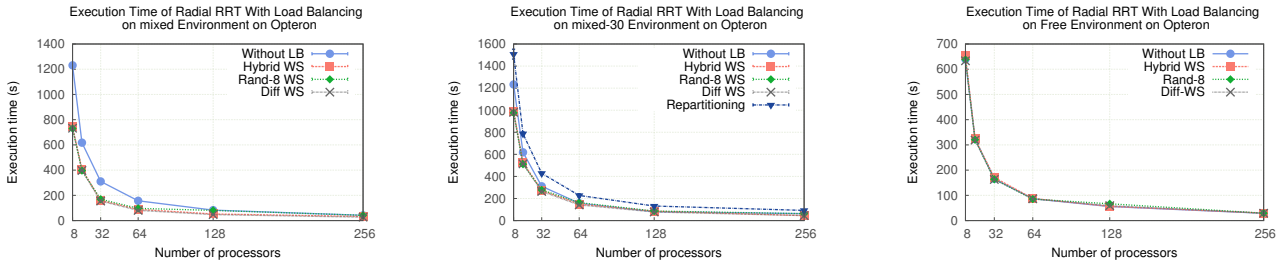

Figure 10. Execution time for RRT with various load balancing strategies in (a) mixed (b) mixed-30 (c) and free environment.

We find that all three work stealing strategies show similar improvements in execution time, as the problem is overdecomposed to such a degree that underloaded processors have a high probability of finding work, regardless of the victim selection. As expected, work stealing shows a larger improvement in the mixed environment vs. the mixed-30 environment, as the reduction in execution time is a function of the amount of imbalance. Similar to the PRM experiments in which we measure load balancing overheads, we find that in the free environment, we do not see significant differences in the load-balanced execution vs. the baseline.

V. RELATED WORK

Parallel Motion Planning. For over three decades, researchers have proposed and studied different types of parallel algorithms for motion planning problems. For a comprehensive survey of early work in parallel motion planning, please see [17]. Recently, research efforts have focused on parallel sampling-based motion planning. These renewed efforts are in part encouraged by the progress made in sequential algorithms, the ubiquity of parallel and distributed machines, and the demand for more efficiency in solving motion planning problems.

In [1], the authors make a case for the "embarrassingly" parallel nature of PRM and present a parallel algorithm in which each processor generates an "equal" number of samples and attempts connection between each sample and its k-nearest neighbors. The original shared-memory algorithm was later applied to protein folding [31] and extended for distributed-memory machines. The major drawback of this approach was the all-to-all communication involved in the global search to find nearest neighbors, limiting scalability to large systems or problem sizes.

Carpin and Pagello [7] propose parallelizing RRT computations on shared-memory machines. Bialkowski et al. [2] provide a parallel GPU-based RRT and RRT* by focusing on parallelizing the collision detection phase. A more recent work focused on multicore architectures [11]. The authors present three algorithms for distributed RRT. The first algorithm is a message passing implementation of the OR parallel paradigm. In the second algorithm, each process builds part of a tree and globally communicates with the other processes each time a new node and edge is added. The third algorithm adopts a manager-worker approach where the manager initializes a single tree, while the expansion computation is delegated to the workers. However, this approach does not scale well as it is prone to load imbalance.

Load Balancing and Work-Stealing. Load balancing in parallel computations is a well-studied problem. Work stealing has become the *de facto* dynamic scheduling technique for various parallel programming environments and runtimes, including Cilk [3], TBB [18], UPC [13] and many others. Blumofe [4] shows that work stealing is provably optimal within a constant factor for scheduling multithreaded computations with dependences. These approaches prove

successful in shared-memory architectures, but have their limitations when applied to distributed-memory. For shared-memory, the issue of locality is generally not stressed. Recently, locality-aware work stealing implementations began placing more emphasis on the notion of affinity [15] and have been shown to perform well in practice.

The X10 programming language [9] and runtime system offers work stealing in distributed-memory architectures. Particularly, X10's lifeline work stealing approach has shown success in balancing load for various applications, including the UTS [26] benchmark. Chapel [6] is a parallel programming language that runs in distributed-memory and provides support in work stealing scheduling.

Charm++ [21] is a parallel programming language and runtime environment that supports a large suite of load balancing mechanisms. In the Charm programming environment, computations are expressed as objects that represent both the work and associated data. In such a model, the work and data are inherently coupled, making it difficult to reason about a data structure or describe a computation in a parametric and data-independent fashion.

In addition to work stealing, other popular approaches for load balancing include partitioning tools for meshes, arbitrary graphs and other data structures. Zoltan [12] and ParMetis [22] are just a few such frameworks that provide repartitioning algorithms and data management tools.

VI. Conclusion

This work reviewed sampling-based motion planning methods in sequential processing and two parallel algorithms based on PRM and RRT. We highlighted load imbalance in these algorithms and presented load balancing techniques to address these issues. This work presented a theoretical model analyzing the efficacy of these load balancing techniques and showed that using repartitioning allows for achieving a distribution of load close to the ideal. In addition, we compared repartitioning and work stealing, finding that work stealing is a suitable approach for all cases, but does not perform as well as repartitioning. In contrast, we showed that repartitioning is extremely effective in balancing load for parallel sampling-based motion planning, but is not applicable to RRT-based algorithms as it is to PRM-based algorithms. Finally, we validated our approach on two massively parallel architectures more than 3,000 cores.

References

[1] N. M. Amato et al. Probabilistic roadmap methods are embarrassingly parallel. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*. 1999.

[2] J. Bialkowski et al. Massively parallelizing the RRT and the RRT*. In *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, 2011.

[3] R. D. Blumofe et al. Cilk: an efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symposium on Princ. and Prac. of Par. Prog*, PPoPP '95, NYC, NY, USA, 1995.

[4] R. D. Blumofe et al. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[5] A. Buss et al. STAPL: Standard template adaptive parallel library. In *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, pages 1–10, New York, NY, USA, 2010. ACM.

[6] D. Callahan et al. The Cascade High Productivity Language. In *Wkshp. on High-Level Par. Prog. Models and Supportive Env.*, volume 26, pages 52–60, Los Alamitos, CA, USA, 2004.

[7] S. Carpin et al. On parallel RRTs for multi-robot systems. In *Proc. Italian Assoc. AI*, pages 834–841, 2002.

[8] H. Chang et al. Assembly maintainability study with motion planning. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*. 1995.

[9] P. Charles et al. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[10] H. Choset et al. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, June 2005.

[11] D. Devaurs et al. Parallelizing RRT on distributed-memory architectures. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2011.

[12] K. Devine et al. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*. 2002.

[13] T. El-Ghazawi et al. UPC: unified parallel C. In *Proc. of ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006.

[14] V. Garcia et al. Fast k nearest neighbor search using GPU. In *CVPR Workshop on Computer Vision on GPU, Anchorage, AK, USA*, 2008.

[15] Y. Guo et al. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 341–342, New York, NY, USA, 2010.

[16] Harshvardhan et al. The STAPL parallel graph library. In *Lang. and Comp. for Par. Comp. (LCPC).*, LNCS. Springer Berlin, 2012.

[17] D. Henrich. Fast motion planning by parallel processing - a review. *Journal of Intelligent and Robotic Systems*, 20(1):45–69, 1997.

[18] Intel. *Reference Manual for Intel Threading Building Blocks,* , 2009.

[19] S. A. Jacobs et al. A scalable method for parallelizing sampling-based motion planning algorithms. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2012.

[20] S. A. Jacobs et al. A scalable distributed RRT for motion planning. In *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2013.

[21] L. V. Kale et al. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[22] G. Karypis et al. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proc. of the 1996 ACM/IEEE Conf. on Supercomputing* , SC '96, Washington, DC, USA, 1996.

[23] L. E. Kavraki et al. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. Automat.*, 12(4):566–580, August 1996.

[24] S. M. LaValle et al. Randomized kinodynamic planning. *Int. J. Robot. Res.*, 20(5):378–400, May 2001.

[25] D. Musser et al. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[26] S. Olivier et al. UTS: an unbalanced tree search benchmark. In *Proc. of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[27] E. Plaku. Distributed computation of the knn graph for large high-dimensional point sets. *J. of Par. and Dist. Comp.*, 67(3), 2007.

[28] J. H. Reif. et al. Complexity of the mover's problem and generalizations. In *Proc. IEEE Symp. Foundations of Computer Science (FOCS)*, pages 421–427, San Juan, Puerto Rico, October 1979.

[29] A. P. Singh et al. A motion planning approach to flexible ligand binding. In *Int. Conf. on Intelligent Systems for Molecular Biology (ISMB)*, pages 252–261, 1999.

[30] G. Song et al. Using motion planning to study protein folding pathways. In *Proc. Int. Conf. Comput. Molecular Biology (RECOMB)*, pages 287–296, 2001.

[31] S. Thomas et al. Parallel protein folding with STAPL. *Concurrency and Computation: Practice and Experience*, 17(14):1643–1656, 2005.