Collaborative Threads: Exposing and Leveraging Dynamic Thread State for Efficient Computation

Kaushik Ravichandran
Georgia Institute of Technology

Romain Cledat
Georgia Institute of Technology

Santosh Pande
Georgia Institute of Technology

Abstract

Current traditional models of parallel computing rely on the static breaking-up of computation or data: multiple parallel threads independently execute different parts of the overall computation with little or no communication between them. Under current models, inter-thread communication is limited to data that carries little semantic information about the *role* of the thread in the overall application. Adding such knowledge and sharing it among threads would allow a *collaborative* model of computation.

In this paper, we present a novel programming model where threads expose their internal *higher order computational state* allowing the construction of a global view of the program's computations, which we call the computational state tree (CST), enabling several optimizations. We discuss how the CST can be used by threads to orient the computation where it would be most useful or to re-use results already computed by other threads.

We present a method to extract collaborative information from the states of threads and insert it into the CST. We implement our method through a runtime and develop an API that can be used to leverage collaboration. We demonstrate how collaboration can be used to orient the computation of a SAT solver to maximize the number of satisfiable assignments found and also how collaboration through results sharing can be used to speedup a K-Means computation.

1 Introduction

Parallel programming today mostly relies on the concept of *threads*. Threads were introduced in the operating systems world to enable spawning off of long blocking-calls mainly involving I/O while still allowing the processor to be utilized fully. Today, threads are the main workhorses of parallel programming, playing a central role in the execution of concurrent computations on multi-core machines. Thread semantics have, however, remained mostly unchanged from the days they were used to achieve multi-tasking on a single processor to their present avatar where they run concurrently on multiple cores. Different programming paradigms effectively utilize threads such as: the producer-consumer paradigm [10], the thread-pool paradigm, etc. More recently, programming models have emerged to create higher abstractions on top of threads: the use of languages such as Cilk [7], X10 [2] or libraries such as Intel's Threading Building Blocks [6] allow the programmer to think in terms of parallel *tasks* and automatically manage the mapping between tasks and threads. STMs [9, 5] have greatly alleviated the problems of deadlocks and race conditions by allowing the programmer to reason at a much higher level of abstraction than pure locks. Models such as the Galois programming model [8] leverage both the tasking model and STMs to offer a very high-level abstraction to "Joe" programmers.

All of these models, however, fail to add expressiveness to threads: a computation running in a thread is unaware of its role in the overall program as well as the role of other concurrent computations. Communication between threads is limited to the sharing of values (shared data communication through shared memory or message passing), and basic synchronization (such as waiting at a barrier) and carry little semantic information about a thread's semantic state. Such lack of knowledge of a thread's computational state and semantics limits its expressiveness.

Parallel computing was traditionally envisioned as the division or splitting-up of work and trying to load balance it as best as possible. In [3], we advocated a new view of leveraging parallelism to overcome sequential bottlenecks by using parallel resources to trigger competing computation and letting the winner be decided dynamically for best speed-up. In this work, we propose another alternative view of parallelism: the use of parallelism to promote *collaboration* amongst threads to better parallelize and steer useful computation. We propose that threads share their *higher-order computational state* information. By this we mean that threads should be made aware of the semantic value of their computation and that of others. This semantic *higher-order computational state* could be representative of several different kinds of information including: the results threads are computing, the role they play in the computation, the insight they may have gained about the problem, how much time peer threads take to execute, etc. Exposing such state information and its subsequent usage allows us to envision a model where a task is not bound to a particular thread but is modified based on the global state of all other concurrent tasks. A task could therefore dynamically change itself based on the work performed by other tasks.

1.1 Contributions

We make the following contributions in this paper:

- We motivate the need and utility of sharing computational state amongst threads.
- We introduce the Computational State Tree, an efficient representation of the semantic computational states of threads.
- We demonstrate a system which enables the use of shared state information to speed-up a SAT problem through the orientation of its computation as well as the K-Means algorithm through result sharing.

The remainder of the paper is organized as follows. In Section 2 we present several examples of computational state and their applications. We then discuss modeling of this state information in Section 3. Our API and its use are described in Section 4. We evaluate two specific instances of sharing higher order semantic state: computation orientation and computational result sharing in Section 5 before concluding with future work in Section 6.

2 Computational state: use-cases

Before discussing the modeling of computational state in Section 3 we motivate a few use cases of computational state through examples.

2.1 Hot/Cold spots: orientation

In certain algorithms where a large space has to be explored, sharing meta-information about the space already explored can be leveraged by sibling threads to navigate away from “cold” spots and towards “hot” spots where solutions are more likely. This is particularly true of SAT problems in the region between the SAT and UNSAT phases [1], which we shall further discuss in Section 4.3.1. As an illustrative example, consider the simple problem of finding a maxima of a complex function in a n-dimensional space. A simple approach could be to concurrently pick randomized starting points and perform a local search from that starting point (such as simulated annealing). As the algorithm progresses, certain areas of the space become “hot”, where the function takes on large values, whereas others become “cold” where smaller values are observed. Therefore, instead of picking random starting points or solely relying on simulated annealing to give the direction of search, this added meta-information about the problem space can be leveraged to *orient* the computation towards what seems to be more profitable areas.

2.2 Computational results: reducing redundancy

Threads can also benefit from sharing certain computational results. Other threads performing *similar* computations can leverage these previously computed results rather than recomputing them from scratch. This is particularly true for combinatorial algorithms which frequently compute but internalize partial results. As an illustrative example, consider the NP-complete sum of subsets problem which is stated as follows: given a set of positive and negative integers, does the sum of some non-empty subset exactly equals zero? The brute force algorithm will iterate over all 2^n subsets of the input data set (of size n). Trivial parallelization can achieve a speedup by dividing and assigning the work among the threads: each subset computation is assigned to one thread. Sharing of results can be very useful here. For example, consider the computation of the sum of $\{1, 2, 3, 4, 5, 6, 8, 9\}$. If some other thread has already computed the sum of $\{1, 2, 3, 4, 5, 6, 7, 8\}$, we can simply re-use this sum by adding $\{9\}$ to it. Similarly, if some other thread has computed $\{2, 3, 4, 5, 6, 8, 9, 10\}$, we can still re-use this information by making minor modifications (adding $\{1\}$ and subtracting $\{10\}$). This example illustrates an important point in our model: the non-specificity of the shared result. Statically, it is not known which sub-result can and will be used and our model accounts for this fact. Our approach is in a way similar to incremental computations or dynamic programming [4, Chap. 15] but is far more flexible and imprecise. In dynamic programming, results to statically determined sub-problems are reused, whereas we use the results of a *similar* and potentially unspecified (at compile time) computation. The specific scheduling order of threads will completely change which results are already computed and therefore change which ones can be reused. This makes our approach much more amenable to multi-threaded environments where any sort of dependency typically induces a slow-down.

3 Modeling

Fundamental to our model is the sharing and utilization of the *computational state*. This Section details how we can efficiently model *computational state*.

3.1 Semantic state

Traditionally, the *state* of a computation C at a given point in time t can be defined as the state of the memory, the registers and the program counter. However, this information is extremely low level and does not give any indication of what C is actually doing for the application as a whole. Take for example a sorting computation on an array A using the quicksort algorithm after the selection of 3 pivot points. The state of the computation, in the traditional sense, would be captured by the state of the array in memory whereas the semantic computational state would be “sort on A having placed 3 elements”. In this trivial example, we can already see that the semantic state must answer the questions: **a)** what computation, **b)** on what data and optionally **c)** what progress the computation has made. The computational state is thus not abstract or difficult to obtain for the programmer, rather it is information that is lost in translation between the design of the program and its actual implementation in code. The exact nature of the *computational state* can vary from program to program. To maintain generality, we simply impose the following two restrictions:

- **Distinguishability** Two different computations should be distinguishable. This rule enforces that two identical computational states represent the same computation from a semantic point of view.

- **Minimality** While the first restriction will push to expand the amount of information encompassed in the computational state, this restriction forces the state to be as small as possible. In other words, two computations that are the same semantically should have the same computational state. Note that this does not mean that the underlying computations are exactly the same. For example, different algorithms could be used to implement the same semantic "sort" in the example above.

3.2 State representation

We adopt a simple `(key, value)` model where `key` is representative of a single computational state. The `key` minimally distinguishes between different computations in a program. The `value` is an arbitrary value that says something about the computation represented by `key`. For example, in the case of orienting the computation (Section 2.1) the `value` would be a measure of the success of the computation and in the case of result sharing (Section 2.2), the `value` would be the result of the computation itself.

3.3 State clustering

To effectively use all the state that has been expressed by threads we first need to construct a system-wide view of this information in the runtime. Since collaboration is most effective amongst computations that are similar we cluster similar computations based on the similarity of the `key` in the `(key, value)` pair. To improve efficiency, we construct a hierarchical representation which we call the `Computational State Tree` or CST for short. The CST is therefore the semantic representation of the state of the entire program up until a certain point in time: it contains all current and past computations that have shared information. We use a hierarchical incremental approximate clustering algorithm to build the CST, for the following reasons:

- **Hierarchical** When presented with a key, quick identification of the cluster to which the key belongs is important. Storing clusters hierarchically allows a logarithmic time lookup in the clusters. Hierarchy also allows us to trade accuracy for speed.
- **Incremental** Keys are generated continually and hence the data set to be clustered (the keys) is not available in its entirety during clustering. An incremental algorithm allows us to update the structure without rebuilding it from scratch.
- **Approximate** The algorithm we use is not guaranteed to form the best clusters at all times. However, it will do this very quickly and allow quick insertion and lookups.

Figure 1 shows a clustering that our model would generate.

4 Basic API

The API to the runtime has two distinct roles: **a)** support the identification and comparison of computational state and **b)** support the sharing of computational state.

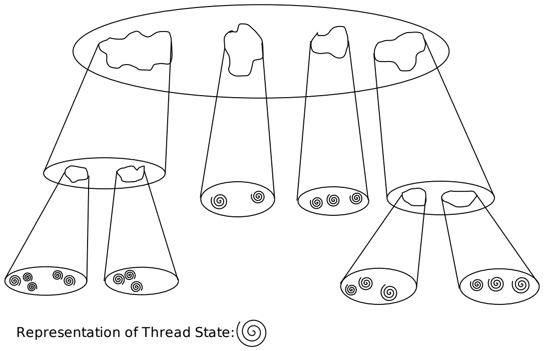

Figure 1: CST: A hierarchical clustering of the states of threads.

4.1 State representation and comparison

A computational state is represented by a `(key, value)` pair as discussed in Section 3.2. We define a `similarity` measure that evaluates how close two computational states are. The `similarity` measure can be viewed as a distance (in the mathematical sense of the term) on the space of computational states. With the help of this programmer defined measure, our system can quickly identify other past/current computations that have/are performing similar computations and leverage it to perform various optimizations. This user-defined function takes as input two `keys` and simply returns a floating point value which is proportional to the distance between the computational states:

```
double similarity(key1, key2);
```

This function needs to be able to compute similarity very quickly. This function is called many times during the operation of the runtime and hence speed is of critical importance.

As an example, consider the sum of subsets problem again. This problem would benefit from the sharing of results between threads as discussed in Section 2.2. Here, the `key`, a minimal representation of the computation, is the set itself. The `value` is the sum computed. For the computation over $\{-1, 3, 6, -10\}$ the `key` would be the set itself and the `value` would be -2. The similarity function would simply compute the cardinality of the *symmetric difference* between the two keys. We can observe that it is extremely simple for the programmer to identify the `key`, `value` and the `similarity` function based on the semantics of the program.

4.2 Computational state sharing

The two most basic primitives that our API provides to support state sharing are:

```
void share(key, value);
{key, value} getClosestState(key);
```

Threads can therefore publish their current state to the runtime through the `share` primitive as well as lookup the closest, most relevant state to the `key` through the primitive `getClosestState(key)`.

4.3 CST usage

We demonstrate how the API can be used to allow threads to *share* their computational state and query the closest computations stored in the CST. We demonstrate the utility of the API in two applications: GSat to demonstrate the hot/cold spot orientation use-case and K-Means to demonstrate the sharing of computational results use-case.

4.3.1 Hot/Cold Spots: Orientation

We demonstrate our approach to hot/cold spots with the GSat SAT Solver. The GSat SAT solver assigns a random truth assignment and then iteratively refines this solution by choosing and flipping a variable which minimizes the number of unsatisfied clauses in the new assignment. This is repeated until a satisfying solution is found or the maximum number of iterations is reached at which point the algorithm fails. We use the GSat SAT Solver as an all-solution SAT solver aimed at finding all satisfying assignments. A trivial parallelization of this problem is to assign different threads with different random truth assignments (in different parts of the solution space) and run them until the entire space is explored.

Opportunity in GSat For a given number of boolean variables, the problem is known to be most difficult when the ratio α of the number of clauses to the number of boolean variables is approximately 4.27 [1] due to the appearance of small pockets of satisfying solutions separated by large distances. Typical solvers like GSat easily get stuck in local minimas. Since we know that solutions are clustered, we use threads which discover satisfying results as indicative of "hot" spots in the solution space. Threads which discover satisfying assignments publish this information into the CST by using the `share` primitive. Their current truth assignment (which is satisfying) serves as the key. For example, the truth assignment $(A = true, B = false, C = true)$ serves as the `key` and the `value` is simply 1 indicating that a solution was found for that `key`. The similarity metric is simply the *Hamming Distance* between two keys in the solution space. The higher the number of threads publishing successes in certain regions of the solution space, the denser those clusters become in the CST. Other threads request the system for close-by "hot" areas by issuing a simple lookup call `getClosestState(key)` where `key` is their current truth assignment. With a probability of p, we migrate threads working in other areas over to these "hot" spots to increase the rate of generation of satisfying assignments. Migration is performed by making minor modifications to the current assignment to move towards the "hot" spots in the solution space. This leads to a quicker rate of discovery of satisfying assignments.

4.3.2 Computational Results: Reducing redundancy

We demonstrate our approach to reusing results through the K-Means algorithm. K-Means is an extremely popular clustering method. It clusters a set of N points into K clusters. Each point is clustered into the cluster which has the closest mean or centroid. The most common algorithm that is used to implement K-Means is an iterative refinement technique referred to as Lloyd's algorithm which we briefly describe. K initial centroids are randomly selected from the N input points. K clusters are then generated by associating each point to its nearest centroid. The centroid is then recalculated for each of these clusters and the algorithm starts again, re-associating points to their closest means. This continues until convergence is reached. K-Means is trivially data-parallel and each thread is in charge of a disjoint set of points.

Opportunity in K-Means The most time consuming stage of Lloyd's algorithm is the stage in which each of the N points' closest centroids are determined. The standard algorithm makes K distance computations for each point (to each of the centroids). As the number of centroids (K) increases, the cost of computing these distances quickly increases.

However, if we observe the operation of the algorithm carefully, we note that a large amount of computation can be reused between the computations for each point. For example, if we compute the closest centroid, say C, for a particular point A, by computing all the K distances to each centroid. A point B, extremely close to point A, will with very high probability have C itself as its closest centroid as well. Even in the case that it is not C it will be one of the closest centroids of A. We use this observation to re-use the closest centroid computation from one point to another.

Using our model, when a thread computes the closest centroid it publishes it into the CST using the `share` primitive. Its current spatial co-ordinates $(x_1, x_2, ..., x_n)$ serve as the `key` to the computation. The similarity metric is simply the *Euclidean Distance* between two keys in the clustering space. The `value` is a list of closest centroids. To generate this list, let us assume that the closest centroid a point found was C_1 at a distance d_1. While computing this closest centroid we also generate a list of its closest centroids within a distance of $d_1 + 2 * T$ where T is a user-defined lookup threshold. This list, $(C_1, C_2, C_3 ... C_n)$ serves as the `value`. Subsequently, when a thread begins a computation for a point it looks for close-by results to see if it can re-use any. By looking up results within a lookup threshold, T, the thread can determine its closest centroid by computing distances only with the centroids in the `value` list instead of all K centroids. This leads to a large reduction in computation time.

5 Experimental results

To validate our work, we have implemented a simple runtime implementing the CST and we have used it to share information on hot/cold spots as well as results across threads. The runtime and examples were all implemented in Java and run with Sun Java 1.6. All experiments were performed on a dual quad-core Intel Xeon E5540 (2.53GHz) with up to 8 concurrent threads.

5.1 Hot/Cold Spots

As explained in Section 4.3.1 we demonstrate our approach to hot/cold spots with the GSat SAT Solver.

Speedups We ran a parallel version of the GSat algorithm over a problem with 38 boolean variables and an α of 4.2. We observed that a larger number of successful assignments are discovered in a given time frame when we enable collaboration using "hot" spot information. Note that we used a low number of variables due to limitations in the Java implementation of GSat used. We ran the parallel versions for 2 minutes each and measured the number of unique satisfying solutions that the solvers were able to find. The solver which was running with collaboration turned off (ie: the original non-collaborative parallel solver) discovered 11 solutions whereas the solver running with collaboration turned on discovered 14 unique solutions, an improvement of over 25%.

5.2 Result sharing

As explained in Section 4.3.2 we demonstrate our approach to result sharing with the popular K-Means clustering algorithm.

Speedups Figure 2 shows the speedup obtained with result sharing for $100,000$ points randomly distributed into K clusters. The speedup is measured by comparing the execution time of the vanilla parallel K-Means solver with that of the same solver using our collaboration framework. We can see that the benefit of sharing is fairly constant and increases dramatically as the number of clusters increases.

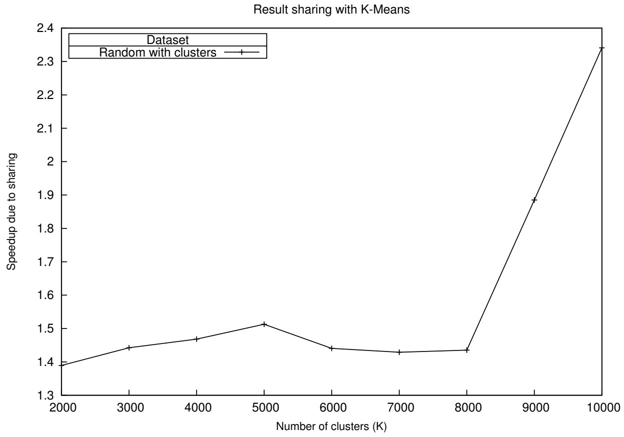

Figure 2: Speedups obtained through results-sharing in K-Means

5.3 Runtime overhead

One of the design goals of the CST was low overhead while inserting and looking elements up. Figure 3 shows both overheads in microseconds as a function of the size of the CST. The measure for the look-up overhead is an average over 1000 random look-ups in a CST of a fixed size. The insertion overhead for X nodes corresponds to the average time to insert $10,000$ elements in a CST of size $X - 10000$. The look-up time smoothly increases logarithmically with the number of nodes in the CST which is consistent with a tree representation. The insertion times are more chaotic as the insertion of an element may cause certain clusters to be split-up and reorganized; however they scale well with the number of elements (150% increase in time for a 2500% increase in CST size).

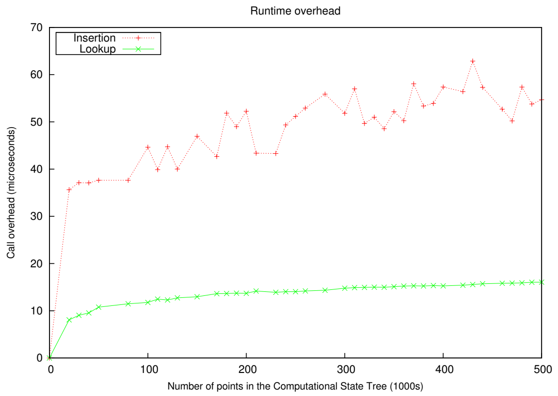

Figure 3: Overhead of insertions and look-ups in the CST

6 Conclusion and future work

In this paper we proposed a new programming model for parallel computing leveraging collaboration between threads through the sharing of their computational state. We motivated the utility of collaboration and demonstrated it in the orientation of the computations and the sharing of results. We believe our proposed representation of the global semantic state of a program through the CST will lead to many interesting collaboration opportunities which we have briefly touched on in this paper.

We are currently working on several other uses of the shared state in the CST including core selection and prioritization of highly requested computations. We are also looking at how best to manage the amount of shared information. Indeed, scalability is very important for our work and since information is only shared and not deleted, we are looking at how best to garbage collect it. Finally we are also looking at ways of better expressing the semantic state of a computation. Currently, we leave it up to the programmer to determine the state to share but would like to make it much more seamless.

More generally, we are also investigating how making more semantic information, of which the CST is an example, available at runtime can improve the performance of applications.

7 Acknowledgments

The authors would like to thank the anonymous reviewers for HotPar whose comments were helpful in finalizing this paper. The authors also gratefully acknowledge the support of NSF grants CCF-0702286 and CCF-0916962.

References

[1] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: an algorithm for satisfiability. *CoRR*, cs.CC/0212002, 2002.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an

object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[3] R. Cledat, T. Kumar, J. Sreeram, and S. Pande. Opportunistic computing: A new paradigm for scalable realism on many cores. In *HotPar 2009: 1st USENIX Workshop on Hot Topics in Parallelism*. USENIX, 2009.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001.

[5] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[6] Intel. Intel threading building blocks. `http://www.threadingbuildingblocks.org`.

[7] C. F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, Jan. 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[8] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07*, pages 211–222, 2007.

[9] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[10] A. S. Tanenbaum. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2001.