RC25256 (W1110-002) October 3, 2011
Computer Science

IBM Research Report

Relaxing Synchronization for Performance and Insight

Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, Daniel Prener, Colin Blundell
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Relaxing Synchronization for Performance and Insight

Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan,
Ravi Nair, Daniel Prener, Colin Blundell[1]
IBM T. J. Watson Research Center
Yorktown Heights, USA
(lrengan, viji, nair, prener)@us.ibm.com, blundell@gmail.com

***Abstract*—Synchronization overhead is a major bottleneck in scaling parallel applications to a large number of cores. This continues to be true in spite of various synchronization-reduction techniques that have been proposed. Previously studied synchronization-reduction techniques tacitly assume that all synchronizations specified in a source program are essential to guarantee quality of the results produced by the program. In this work we examine the validity of this assumption by studying the effect of systematically relaxing synchronization for a wide set of parallel programs.**

We consider the class of computations for which the quality of result can be quantified, and the quality need only fall within some acceptable range. Several computations from important benchmark suites such as PARSEC, STAMP, and NU-MineBench, belong to this class. We propose a technique which relaxes the programmer-specified synchronization to reduce, and in some cases completely eliminate, the synchronization overhead. In addition, we also develop model computations that map to some of the well-known parallel applications, and analyze the effects of relaxing synchronization. Our results show that relaxing synchronization can achieve significant speedups; for example, up to 15x for the *Kmeans* benchmark and up to 70x for one of the model computations, with no degradation in the quality of the results. Our study also provides valuable insight into the synchronization resilience of parallel applications, thus paving the way towards the development of more effective synchronization primitives that can be exploited efficiently by compilers and parallel runtime systems.

I. Introduction

Parallel computer systems are gaining in importance largely because of the need to process the vast amount of data that is being produced from all types of computing devices and sensors. These systems are also being called upon to perform complex analysis of data and to answer queries from millions of users in real time. As systems get larger however, they get more complex and hence more expensive and more power-hungry. Both the acquisition cost and the running cost of computers can be contained by recognizing that there is a precision implied by traditional computing that is not needed in the processing of most new types of data. This relaxation of precision can help in the wider exploitation of relatively more energy-efficient modes of computing like cluster computing. More importantly, this relaxation of the requirement of deterministic execution is in tune with the trend in parallel systems designed for computing results of approximate nature.

[1]The work presented in this report was performed while Dr. Blundell was affilitated with IBM Research. The author is currently affliated with Google.

One source of both hardware complexity and software overhead in most modern parallel systems is the act of synchronization between parallel threads. Such synchronization may occur either in order to ensure that fundamental data structures, e.g. linked lists, do not break due to simultaneous manipulation of their structure by different threads, or in order to ensure that threads reach various points in their execution in a predictable manner, or in order to ensure that all threads see consistent values when shared variables are updated. While the first of these reasons for synchronization cannot typically be relaxed because it could lead to a fatal crash of the program, the second, and even more often, the third reason can be relaxed with no visible effect in many cases and acceptable effect in other cases.

Traditional synchronization techniques like data privatization [2][11], lock-free synchronization [10][8][1] and transactional systems [8][14] have helped a lot in reducing synchronization overheads. These techniques primarily use careful analysis and identification of situations where synchronization is not needed, or use hardware or software structures that are able to detect and correct incorrect behavior due to relaxed synchronization. However this has added complexity along other dimensions, while not fundamentally alleviating the barrier that synchronization poses to the scaling of parallel applications. We are therefore led to ask what the effect would be if synchronization were omitted in situations where such an omission would not lead to a catastrophic crash of the system. Would the program produce results that are *acceptable*? Furthermore, if the results are acceptable, would they be produced in considerably less time?

This is what we set out to explore in this work. We have taken a wide range of programs from areas that tolerate some imprecision in the results, and show that a large percentage of them can tolerate relaxation of synchronization. Our results are similar in spirit to those of Rinard et al. [13] who show that there are classes of problems where the omission of certain operations, or even of certain loop iterations in their entirety could speed up programs while producing results that are not accurate but still acceptable.

Rinard et al. also pioneered the notion of model computations to identify classes of single-threaded applications that lend themselves to performance speedup while sacrificing quality of results to an acceptable extent. We do the same for parallel programs, developing model

computations for such programs, and examining speedup through relaxation of synchronization. Our results suggest that even when strict synchronization is not enforced, the probability of concurrent modification of a shared variable by more than one thread is low. Even when updates to shared variables do collide, the updated values are not different enough to cause the results to deviate too far from what they would have been in the presence of synchronization.

We develop a synchronization relaxation technique which reduces, and in some cases completely eliminate, synchronization overhead. Our study provides insight into the characteristics and behavior of various classes of parallel applications when programmer-indicated synchronization in these programs is relaxed using our technique. In addition, we also develop model computations that map to some of the well-known parallel applications, and analyze the effects of relaxing synchronization. Understanding the resilience of applications under such relaxation can help compiler writers to develop automatic techniques to reduce synchronization overhead. It can also help architects design better, and possibly more lightweight, synchronization primitives.

Our results using relaxed synchronization on a wide variety of parallel applications and model computations show that, as we increase the number of threads, and consequently the contention for shared updates, we are able to achieve significant speedups, up to 70x for one of our model computation, and up to 15x for the Kmeans clustering benchmark. Furthermore, our results show that in almost all the cases, we do not have to trade-off the quality of results to improve the execution time, confirming that the mutual exclusion ensured through synchronization essentially comes for free using our approach.

The rest of the paper is organized as follows: In Section II we provide a background for our work with a description of the synchronization problem, and describe the way in which we relax synchronization for various types of workload. Section III describes the experimental methodology used for our study. In Sections IV and V, we present the results that we observed along with an analysis. In Section VI we contrast our approach to related work done by other researchers. Section VII presents our conclusions from this work and our ideas for further work.

II. Relaxed Synchronization

Synchronization overhead is often a major performance limiting factor in parallel applications. For example, Figure 1 shows the parallel execution time of the *Kmeans* [12] benchmark run on a IBM Power P7 machine using the supplied *large* input set with the goal of finding 8 clusters. Clearly, the synchronization time dominates the total execution time and is as high as 90% for 8 threads. We propose here a technique called *relaxed synchronization* that seeks to reduce (and in some cases eliminate) the synchronization overhead.

We describe *relaxed synchronization* using an illustrative example. Consider the computation of arithmetic

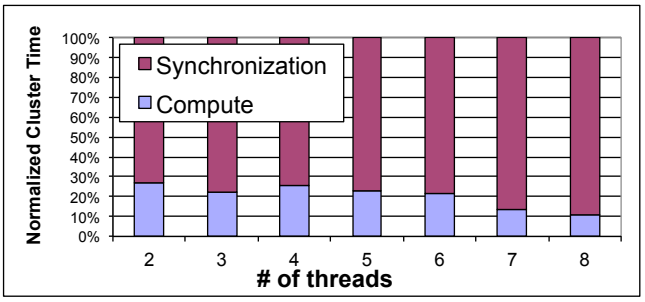


Figure 1: Synchronization overhead in OpenMP based Kmeans benchmark from NU-MineBench

```
double Mean(double *numbers, long n)
{
    double sum = 0.0;
    long i, num = 0;
    #pragma omp parallel for private(i) \
     shared(numbers, sum, num, n) schedule(static)
    for (i=0; i<n; i++) {
      #pragma omp critical
      {
        sum += numbers[i];
        num++;
      }
    } // end for-i
    double mean = sum/num;
    return mean;
}
```

(a) Model computation Mean.

```
double Mean_Relaxed(double *numbers, long n)
{
    double sum = 0.0;
    long i, num = 0;
    #pragma omp parallel for private(i) \
     shared(numbers, sum, num, n) schedule(static)
    for (i=0; i<n; i++) {
      if (i % RELAX_FACTOR == 0) {
          sum += numbers[i];
          num++;
      } else {
        #pragma omp critical
        {
          sum += numbers[i];
          num++;
        }
      }
    } // end for-i
    double mean = sum/num;
    return mean;
}
```

(b) Mean computation with relaxed synchronization.

Figure 2: Relaxed mean code

mean of n numbers. The basic parallel version of this computation is shown in Figure 2(a). The threads synchronize their accesses to the shared variables `sum` and `num` through the critical section. A relaxed version of the code is shown in Figure 2(b). In the relaxed version, the synchronization is skipped for iterations of the `i`-loop that are a multiple of `RELAX_FACTOR`. By choosing appropriate values for `RELAX_FACTOR`, we can control the degree of relaxation. For example, we can skip synchronization for every iteration by setting `RELAX_FACTOR` to 1, and enforce synchronization for all iterations by setting `RELAX_FACTOR` to $n + 1$. Our quantitative results with relaxed synchronization, shown in Section IV(A), demonstrate that we can completely remove the critical section in this code, and still compute results with negligible error.

In a typical parallel execution different threads that

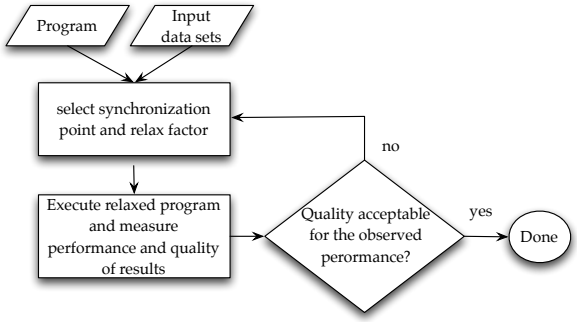


Figure 3: Using relaxed synchronization during program development phase to choose profitable synchronization points and quality-preserving relax factors.

access a shared variable frequently access and update the variable at different times. Thus a significant number of mutual-exclusions are obtained for free even without any synchronization. There will still be cases though where multiple threads update a shared variable concurrently and cause an incorrect update. For some types of computations these incorrect values in the intermediate results do not significantly affect the quality of the final results. Our scheme seeks to exploit the ability of these applications to tolerate approximate results and to execute parallel versions of them with relaxed synchronization.

A. Framework for using relaxed synchronization

We envision that the application developer would relax the synchronizations in the input program and measure the performance and quality of results for representative input data sets during the program development phase as depicted in Figure 3. The goal of this exercise would be to select the appropriate set of synchronization points at which relaxation is safe and to select relax factors that result in faster execution time while producing results with acceptable quality.

Consider a function `f(input)` to which we seek to apply relaxed synchronization. Once a set of profitable synchronization points and relax factors are determined, it is possible to create two versions, `f_relaxed(input)` and `f_original(input)`, which respectively correspond to the relaxed and original version of `f()`. As illustrated in the code snippet below, on a per call basis, the application can choose to use either one of the versions, or if needed both.

```
result = f_relaxed(input);
if (qualityNotAcceptable(result)) {
 // fall back option
 result = f_original(input);
}
```

We expect that for most inputs the relaxed version will produce acceptable results and hence the application will benefit from improved performance. There could be some inputs for which the relaxed version may not produce acceptable results, and in such cases the application can fall back to the original version to compute the exact result. It is important to note that this dual version code allows the application to always compute an acceptable result. However this could occasionally cause an increase in execution time if the relaxed version produces unacceptable results. Thus the worst case execution time of the computation is 2x that of the original. In most cases a properly designed relaxed version should provide adequate approximate results and significant speedups.

One can envision automating the iterative selection of the profitable synchronization points and relax factors, given a program and set of input data sets. For a given choice of the synchronization points and relax factors, the dual version code can be generated automatically. Eventually it should also be possible to use the acceptability criterion dynamically from a running program and adapt the relax factor. We plan to pursue these as future work.

B. Choosing synchronization points

In general, the use of synchronization in parallel programs can be classified into three broad categories: (i) to preserve the integrity of data structures (e.g., malloc(), and insert/delete of nodes in a tree); (ii) for barrier-style collective synchronization; (iii) to ensure that threads use the most recent value of a variable. The first category of structural integrity preserving synchronizations are absolutely essential and do not lend themselves to relaxation. We have indeed observed the abnormal termination of applications when synchronization was relaxed indiscriminately. The second category of collective synchronizations, such as barriers, are good candidates for relaxation, and examples of such relaxation are shown in [9] for the NAS parallel benchmarks using OpenMP directives. The third category of synchronizations that ensure the flow of most recent values to the threads are excellent candidates for relaxation, and this is the class we explore in this work.

III. Methodology

We systematically explore the trade-off between execution time and the quality of results for different classes of parallel applications. For any given application, we build the following three versions of the program binary.

- original – The unmodified version of the program.
- relaxed – The version with synchronization relaxation based on our proposal. The degree of relaxation is determined using a parameter, called *relax factor*.
- reduced – The version using loop perforation [13] as a form of reduced resource computation. As shown in [13], we implement loop perforation by enabling execution of only a subset of the original number of iterations of the parallel loops in an application.

To demonstrate the benefits of relaxing synchronization, our experiments cover a broad set of computation classes with two different parallelism models on two major multicore systems. Table I summarizes the various attributes of our exploration. We explore two major shared memory architecture/platforms: Intel x86 32-way machine and IBM Power P7 32-way machine. We consider two different classes of parallelism / synchronization: explicit via OpenMP and implicit via transactional memory (STM).

Table I: Dimensions explored in our study

Dimension	Attributes
Hardware platforms	32-way Intel (4 8-core Xeon) and 32-way IBM Power P7 (4 8-cores)
Parallel runtimes	OpenMP and STM
Synchronization primitives	OpenMP: Atomic & Critical sections and STM: TM_READ, TM_WRITE
Computation classes	Data mining, graph analysis, image processing, scientific and non-numeric
Compilers	GCC 4.1.2 and IBM XL C 11.1

In addition to using benchmarks from two different suites (NUMineBench [12], STAMP [6]), we develop model computations similar to the models in [13]. These models capture the key computational loops of complex applications, and allow us to study the impact of relaxing synchronization on these parallel loops. The model computations help in gaining insights about program structures which are more amenable to synchronization relaxation, and about the corresponding trade-off in the quality of results. The benchmarks and model computations together cover a variety of computation classes (data mining, graph analysis, scientific, image processing, and non-numeric). For the model computations, we study the impact of reduced resource computation, and compare it with the trade-offs observed with relaxed synchronization. For the benchmarks, we present only the impact of relaxed synchronization.

Table II lists the set of benchmarks and model computations used in our study. Among the benchmarks in the STAMP [6] suite, we were not able to run *Bayes* on the Intel platform. For applications such as *Vacation*, *Genome*, and *Intruder*, the writes from transactions primarily involved modifying a data structure (for example, adding or removing an element from a linked list or queue), or memory allocation/deallocation for pointer. So we did not consider them as candidates for relaxing synchronization. The STAMP benchmarks were compiled and run on the x86 platform due to the availability of STM.

The model computations, *Mean*, *MinSumSet*, and *Sum* are parallel versions which we developed based on the corresponding sequential versions in [13]. We have also developed two additional model computations, *RandCondUpdate* and *RandHistogram* which we describe in the results section. The benchmarks from NUMineBench and the model computations use OpenMP and were compiled and run on the IBM Power P7 platform.

IV. Results for Model computations

We first present the results for the model computations adapted from [13], using the same metrics as [13] for performance and quality. Next we present the results for the two new model computations that we developed.

A. Model Mean

The model computation *Mean* calculates the arithmetic mean of a set of n double values. The parallel code that implements this model is shown in Figure 2(a). Rinard et al. [13] also studied arithmetic mean computation as one

Table II: Benchmarks and Model Computations studied

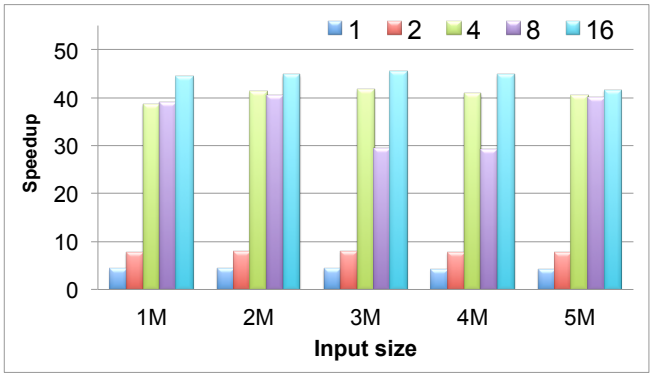
Benchmark/ Model name	Benchmark/Model	Computation class
Kmeans	NUMineBench	Data mining
Fuzzy Kmeans	NUMineBench	Data mining
SSCA2	STAMP	Graph processing
YADA	STAMP	Mesh refinement
Labyrinth	STAMP	Maze routing
Kmeans	STAMP	Data mining
Mean	Model [13]	Scientific computation
MinSumSet	Model [13]	H.264 video encoder
Sum	Model [13]	Scientific computation
RandHistogram	RelSync Model	Data mining, scientific
RandCondUpdate	RelSync Model	Data mining, non-numeric

of his models for reduced resource computing. This model is simple and it provides a good vehicle for exploring the trade-offs between performance and quality of result when relaxing synchronization. It also provides insights into the nature of the relaxed synchronization computations. As observed in [13] arithmetic mean represents a style of reduction computation that occurs commonly in many scientific and data mining computations.

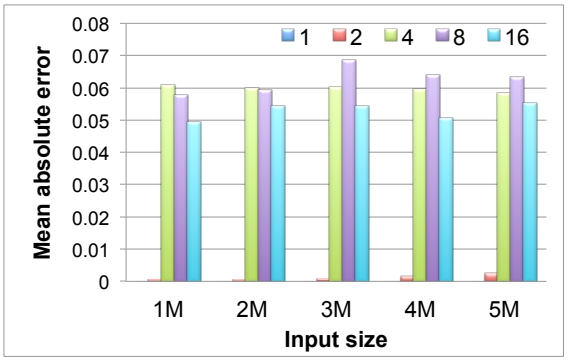
The critical section in the code guards parallel updates to the shared variables `sum` and `num`. The critical section is typically realized using a lock and hence the parallel execution time of this computation is dominated by the synchronization overhead of this lock. We control the relaxation of synchronization by not atomically updating `sum` and `num` variables for some iterations of the parallel `i` loop. Full relaxation of synchronization is achieved when all updates to `sum` and `num` are unsynchronized. This is essentially equivalent to removing the critical section completely. The relaxed version of this code is shown in Figure 2(b).

Figures 4(a) and 4(b) show respectively the speedup and quality of the results for a set of input sizes and thread counts assuming full relaxation of synchronization. All results are averages over 100 trials. The input size n ranges from 1 million to 5 million, and the number of threads varies from 1 through 16 in powers of two. From Figure 4(a) we see that the speedup increases significantly with increase in the number of threads. The synchronization overhead of the original version increases with the number of threads and hence the observed speedup with full relaxation grows as we increase the number of threads for a given input size. This, of course, is accompanied by a degradation in the quality of the result. We estimate the quality by computing the average of the absolute difference between the values computed by the original code and the relaxed version of the code. Figure 4(b) shows the quality of the results for various input sizes and thread counts. Unsynchronized updates to a single variable by multiple threads has a higher probability of causing data races, hence resulting in loss of the effect of one or more updates from multiple threads.

By comparing the speedup and quality in Figure 4 we note that for 2 threads up to a factor of 8 speedup can



(a) Speedup for model computation Mean measured as the ratio of original to fully relaxed parallel execution times.



(b) Quality measured as the difference between the mean computed by the original and fully relaxed versions. The Y axis shows the mean of the absolute differences between the results computed by the original and relaxed version. The smaller the mean absolute error the better the quality.

Figure 4: Speedup and quality of results for the relaxed model computation Mean. Each group of bars corresponds to a particular input size; the bars within a group correspond to number of threads.

```
double Sum(double *numbers, long n)
{
    double sum = 0.0;
    long i;
    #pragma omp parallel for private(i) \
     shared(numbers, sum, n) schedule(static)
    for (i=0; i<n; i++) {
      #pragma omp atomic
          sum += numbers[i];
    }
    return sum;
}
```

Figure 5: Model computation Sum

be obtained with just a negligible error of 0.001 in the computed result. Higher speedups of up to a factor of 45 can be obtained if the user of the results can tolerate an error of at most 0.06, which is 12% of the original result. These results clearly demonstrate the trade-off between performance and quality that the user can dynamically exploit to achieve significant performance improvement with acceptable quality of results. The speedups of 8x for 2 threads and 45x for 4 threads stem from the significant synchronization overhead present in the original parallel version and completely absent in the relaxed version.

B. Model Sum

The model computation *Sum* calculates the sum of n values. The basic parallel code is shown in Figure 5. Rinard et al. [13] also studied the sum computation as one of the models for reduced resource computation. This style of reduction occurs commonly in many scientific and data mining computations as observed in [13].

As with the *Mean* model computation, we experiment with full relaxation for the *Sum* model as well. As before, we observe significant speedup with increase in the number of threads for a given input size. In this case, the quality of the result needs to be defined carefully since the relaxed version may compute a sum with a reduced number of values. As in [13], we estimate the number of values contributing to `sum` variable using a scheme similar to that in the *Mean* model computation. Using this, we

```
long MinSumSet(double **numbers, long numSets,
                        long numElementsInSet) {
  double *sum=(double*)calloc(numSets,sizeof(double));
  double min = DBL_MAX;
  long index = 0;
  #pragma omp parallel for private(i) \
   shared(sum,min,index,numbers,numSets,\
   numElementsInSet) schedule(static)
  for (long i=0; i<numSets; i++) {
    // compute sum of a set
    sum[i] = 0;
    for (long j=0; j<numElementsInSet; j++) {
      sum[i] += numbers[i][j];
    }
    // atomically update global min & index
    #pragma omp critical
    {
      if (sum[i] < min) {
        min = sum[i];
        index = i;
      }
    } // end critical
  }// end for i
  return index;
}
```

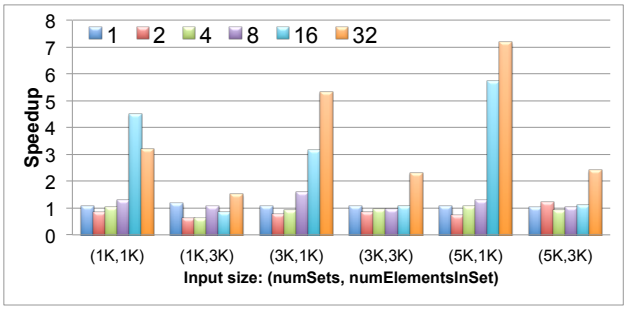
Figure 6: Model computation Minimum Sum Set

compute the approximate sum by adding an estimate of the lost updates. Our speedup and quality of results show trends similar to the *Mean* model computation, and so we omit showing these figures.

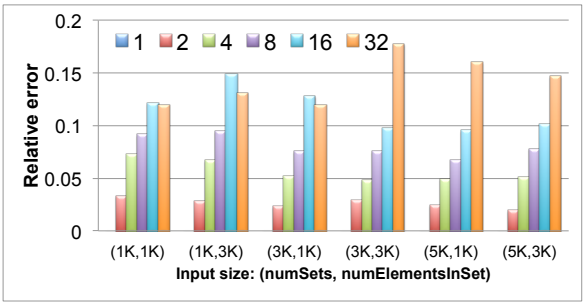
C. Model Minimum Sum Set

The model computation minimum sum set, referred to as *MinSumSet*, finds the set with the minimum sum among a given list of sets. The parallel version is shown in Figure 6, and is derived from the sequential version described by Rinard [13]. As pointed out in [13], *MinSumSet* captures the essence of the technique used for finding similar blocks in an image in the H.264 video encoder in the PARSEC [5] benchmark suite. As shown in Figure 6, each thread is assigned a few sets. In parallel, each thread computes the sum of each set assigned to it, and compares this sum to a global minimum sum to determine whether this set should become the current minimum sum set.

Figures 7(a) and (b) respectively show the speedup and quality of results of the relaxed *MinSumSet* model. The original version is the code shown in Figure 6. The fully



(a) Speedup for MinSumSet model computation measured as the ratio of the original to fully relaxed parallel execution times.



(b) Quality is measured using the relative error of the result computed by the relaxed version to the original. The smaller the relative error the better the quality.

Figure 7: Speedup and quality for MinSumSet model. Input size as pairs of number of sets and number of elements in each set are shown in X axis. Each group of bars correspond to a particular input size and the bars within a group correspond to number of threads.

```
void RandHistogram(int *sumArray,
                   int *indexArray, int n)
{
    #pragma omp parallel for private(i) \
     shared(sumArray, indexArray, n) schedule(static)
    for (int i=0; i<n; i++) {
       #pragma omp atomic
            sumArray[indexArray[i]] += 1;
    }
}
```

Figure 8: Model computation RandHistogram

relaxed version removes the critical sections altogether. The quality of the result is computed using the formula

$$RelativeError = abs(original - relaxed)/(sumRange)$$

where $sumRange$ is difference between the maximum and minimum sum value for a given input set of sets. For example, a relative error value of 0.1 implies that the value computed by the relaxed version is within 10% of true minimum sum value. Figure 7(a) shows that we obtain only modest speedup until the number of threads increases to 16 or more. With fewer threads, the synchronization overhead to update the minimum sum set is relatively less than the time to compute the sum of each set. However, the relative error is greater than 10% for higher thread counts. In the application, H.264 video encoder, if the quality condition for finding similar blocks can be varied depending on the context in which it is being used, our results demonstrate that considerable improvement in execution time can be achieved by relaxing synchronization.

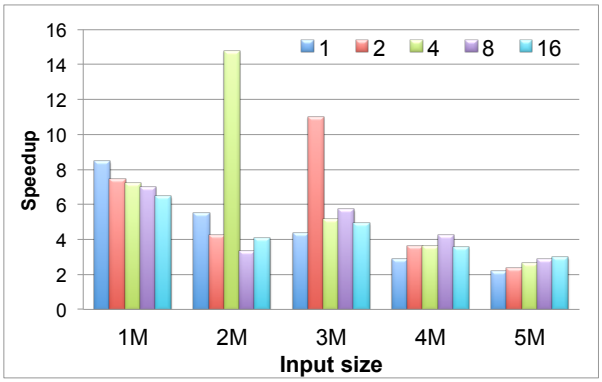
From Figure 7(a), we observe that the speedup relative to the original version drops as we increase the number of elements per set for a given number of sets. As the number of elements per set increases, the overhead of synchronization decreases as more time is spent in computing the sum of the elements of each set.

D. Model Random Histogram

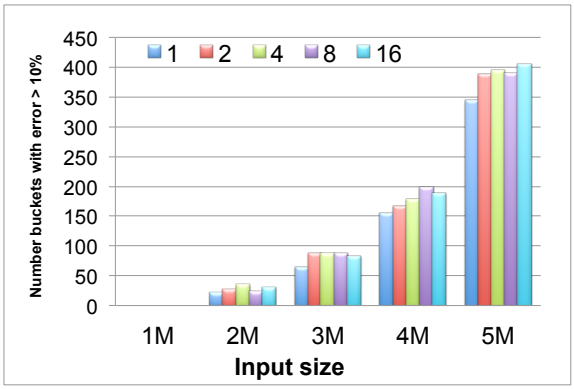
The model computation Random Histogram, referred to as *RandHistogram*, computes the histogram of a set of n values using k bins, where k is also an input to the model. The parallel version is shown in Figure 8. *RandHistogram* represents the computations found in data mining and scientific computations. For example, the *Kmeans* benchmark from NUMineBench [12] has a kernel similar to the computation in the *RandHistogram* model, that computes the number of points assigned to a given cluster, in which n is the number of points and k is the number of clusters. The *RandHistogram* style of reduction with indirect indexing is also found in many scientific computations such as N-body methods. We have identified and developed this model computation to study the effects of relaxing synchronization for this broad class of data mining and scientific computations.

For our experiments, we use a set of values for n, ranging from 1 million to 5 million, and we arbitrarily set the number of bins, k, to be $n/10$. The n numbers are chosen from a uniform random distribution of values from 0 to $k-1$. From the parallel code of the *RandHistogram* model shown in Figure 8, we can see that the atomic update of `sumArray[indexArray[i]]` synchronizes the concurrent updates of the same bin in `sumArray`. Figures 9(a) and 9(b) show speedup and quality of the relaxed *RandHistogram* computation. All results are averages over 100 trials and use full relaxation, i.e. no synchronization of updates to `sumArray[indexArray[i]]`. We see a range of speedup values from 3x to 15x with the speedup decreasing with increased input size. We quantify the quality of the results based on the error in the values computed for each bin. The error in the values computed for each bin is the absolute difference between the values in the original and relaxed versions. For purposes of illustration, we have chosen to show the number of bins whose error is greater than 10% of the original. Figure 9(b) shows the quality of the results for range of input sizes and thread counts. We observe that less than 0.1% of the bins have an error greater than 10% for all input sizes and thread counts. So, essentially we can gain all the performance speedups shown in Figure 9(a) without sacrificing quality in the results.

The speedup comes from the reduced (or eliminated) synchronization overhead in the relaxed version. The negligible loss of quality is due to the following property of the relaxed execution. During a parallel execution a



(a) Speedup measured as the ratio of the original to relaxed parallel execution times.



(b) Quality is measured by comparing the values of bins between original and the relaxed versions. The Y axis shows the number of bins for which the difference between the values of original and relaxed is greater than 10%. Note that the number of bins for a given input size n is $n/10$. Hence, overall we see that there are a negligible number ($< 0.1\%$) of bins that have an error greater than 10% of the original.

Figure 9: Speedup and quality for the model computation RandHistogram. The group of bars in the X axis correspond to different input sizes; bars within a group correspond to different thread counts used in the parallel execution.

```
void RandCondUpdate(int *flagArray,
                    int *indexArray, int n) {
  int i;
  #pragma omp parallel for private(i) \
   shared(flagArray, indexArray, n) schedule(static)
  for (i=0; i<n; i++) {
    #pragma omp critical
    {
      if(flagArray[indexArray[i]] == NOT_TAKEN) {
        // mark element flagArray[indexArray[i]]
       // as taken by current thread
       flagArray[indexArray[i]]=omp_get_thread_num();
      }
    }// end omp critical
  } // end for i
}
```

Figure 10: Model computation Random Conditional Update

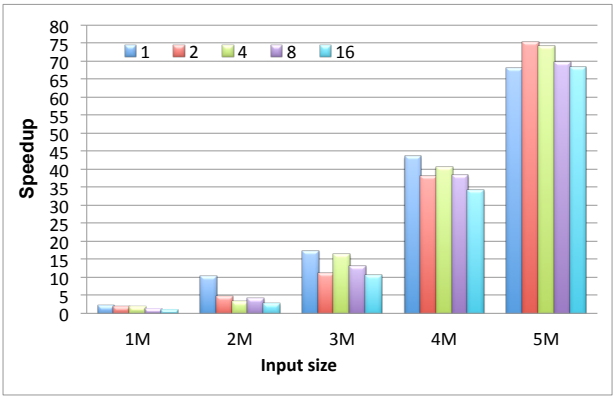


Figure 11: Speedup for the model computation RandCondUpdate. Speedup measured as the ratio of original to relaxed parallel execution times.

significant number of synchronization enforced mutual-exclusions are achieved by the non-overlapping order in which threads access the shared bins. Hence, even when the synchronization is relaxed we still get a good number of mutual-exclusions for free.

E. Model Random Conditional Update

The random conditional update model, referred to as *RandCondUpdate*, implements a computation that conditionally sets a global array of flags. The parallel code is shown in Figure 10. This model represents computations found in data mining and in non-numeric applications like *Labyrinth* benchmark from the STAMP [6] suite. The *Labyrinth* benchmark has a kernel that computes a set of routes through a maze with no two routes being allowed to intersect at a node. Essentially, if a node is taken by one of the routes then it cannot be taken by any other route. This model also serves as a useful vehicle for studying a fundamental question in relaxing synchronization, viz., in a relaxed parallel execution, how many of the synchronization-enforced mutual exclusions were achieved for free and how many of them were violated.

The *RandCondUpdate* model as described in Figure 10 uses the `indexArray` to induce conflicting accesses among different iterations of the parallel `i` loop. In our experiments, we generate random values for the `indexArray` in such a way that it has a given percentage of repetitions. For illustration we have chosen 20% repetitions. Hence, in a parallel execution of input size n there are $0.2n$ potential concurrent updates to a given element of `flagArray`. Without synchronization there are potentially $0.2n$ incorrect updates of the `flagArray`.

We use full relaxation, i.e., all parallel updates to `flagArray` are unsynchronized. Figure 11 shows the speedup obtained for relaxed parallel execution. All results are averages over 100 trials. The speedup is measured as the ratio of the original to relaxed execution times. For smaller input sizes, we see modest performance improvement and as the input size grows we see good to excellent (up to 75x) speedups. To measure the quality we used an additional array to record the elements of

`flagArray` set by each thread. With this information, we later compare the elements of `flagArray` set by each thread to find conflicts – elements that are incorrectly set by more than one thread. Our results show that the actual number of conflicts is negligible. For example, we observe just one conflict out of the $200,000$ potential conflicts for an input of size 1 million. We observed a similar number of conflicts for larger values of n. This result clearly demonstrates that in a typical parallel execution of models such as *RandCondUpdate*, we can get a large number of the synchronizations for free, i.e., without sacrificing the quality of results.

F. Comparison of relaxed synchronization with Rinard's reduced resource computation

Rinard et al. [13] proposed reduced-resource computation as a technique for computing approximate results and evaluated it using a set of model computations (three of which are used in this paper, viz., *Mean*, *Sum*, and *MinSumSet*). The particular reduced-resource computation technique applicable to the models we consider is called *loop perforation*. For a perforated loop only a subset of the iterations of a loop are executed. The perforation factor determines the fraction of iterations executed. For example, a perforation factor of 2 would result in executing only half of the iterations. In this section we compare the potential of our relaxed synchronization technique with that of the loop-perforation technique in terms of performance improvement and quality of results.

Figures 13 shows a comparison for the four models *Mean*, *MinSumSet*, *RandHistogram* and *RandCondUpdate*. The experiments use a parallel version with fully relaxed synchronization and a perforation factor of 4 for loop perforation. It is important to note that the quality of results for loop perforation remains unaffected by the increase in the number of threads because an iteration, if executed, is done with synchronization in place.

For the *Mean* and *MinSumSet* models, the speedup with loop perforation is modest, but with a strong quality of results. On the other hand, for relaxed synchronization there is a spectrum of performance potential and quality of results, giving the user more opportunities to trade quality for performance. For the *RandHistogram* and *RandCondUpdate* models, we see that the relaxed synchronization scheme provides excellent speedup potential and quality of results. On the other hand, the loop perforation technique provides modest performance speedup and results in significant loss of quality.

The comparison results presented in this section cover only one set of input values for the relaxation and loop perforation factors, and is intended to serve as a motivation for further exploration.

V. Experimental Results for the Benchmarks

In this section we study the performance potential of relaxing synchronization in benchmarks from the STAMP [6] and NU-MineBench [12] suite. For all benchmarks we use the large input data sets supplied with the benchmarks and use the quality criteria specified in the benchmark code.

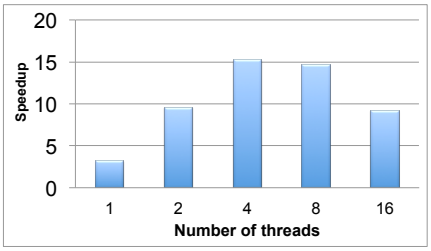


Figure 12: Kmeans from STAMP benchmark suite. Speedup computed as the ratio of the original to relaxed execution times.

We relax the transactional memory (TM) benchmarks from STAMP by replacing one or more of the TM_SHARED_READ and TM_SHARED_WRITE with normal unsynchronized reads and writes. We relax the OpenMP benchmarks from NU-MineBench by removing one or more of the atomic or critical sections and allowing unsynchronized read and writes to shared variables.

All the synchronization primitives which were relaxed in these benchmarks use full relaxation, i.e., all dynamic instances of synchronized updates for that primitive are now performed without synchronization.

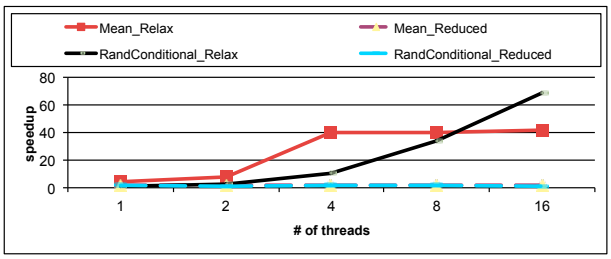
A. Kmeans (STAMP)

The Kmeans benchmark is a software-TM-based implementation of the Kmeans clustering algorithm widely used in data mining. Given a set of n objects, it clusters them into a set of k clusters. A transaction is used to protect the update of cluster centers and is a source for conflicts. The number of conflicts is related to the number of clusters k, the smaller the number of clusters, the greater the number of conflicts.

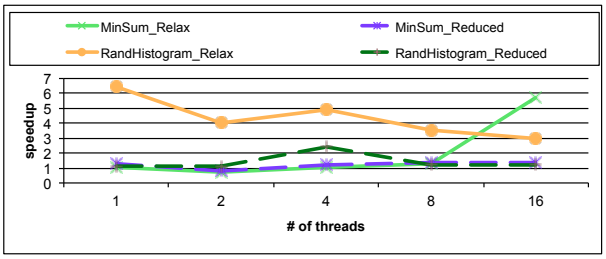
Figure 12 shows the speedup computed as the ratio of the original to relaxed execution times. We observe up to 15x speedups for the large input data set. The code region for which we have relaxed the synchronization is similar to the *RandHistogram* model and a similar analysis applies. The benchmark has an explicit condition for the quality of the result (cluster centers) produced. For the relaxed version we used the same quality condition and all the runs produced results that passed the quality condition. In summary, the results demonstrate that we can obtain large speedups through relaxed synchronization for the Kmeans benchmark with virtually no effect on the quality of the results.

B. Kmeans (NUMineBench)

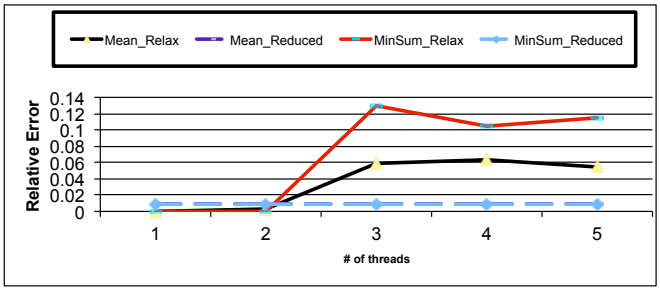
Having seen excellent performance speedup and quality of results for the STAMP TM-based Kmeans computation, we wondered whether the achieved speedup stems from the TM-based synchronization, or whether the Kmeans computation itself is resilient to relaxing synchronization. To explore this, we applied relaxed synchronization to the OpenMP version of the Kmeans computation available in the NU-MineBench. Like the TM version, the Kmeans



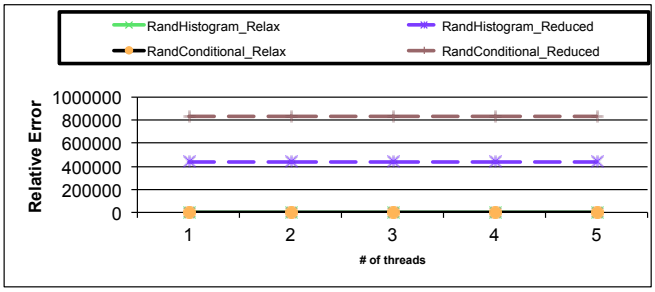
(a) Speedup comparison: Mean and RandCondUpdate



(b) Speedup comparison: MinSumSet and RandHistogram



(c) Quality comparison: Mean and MinSumSet



(d) Quality comparison: RandHistogram and RandCondUpdate

Figure 13: Speedup and quality comparison of the relaxed synchronization and reduced resource computation techniques. Due to the difference in the scales of the results, the grouping of the models in the speedup graphs is different than the grouping in the quality graphs.

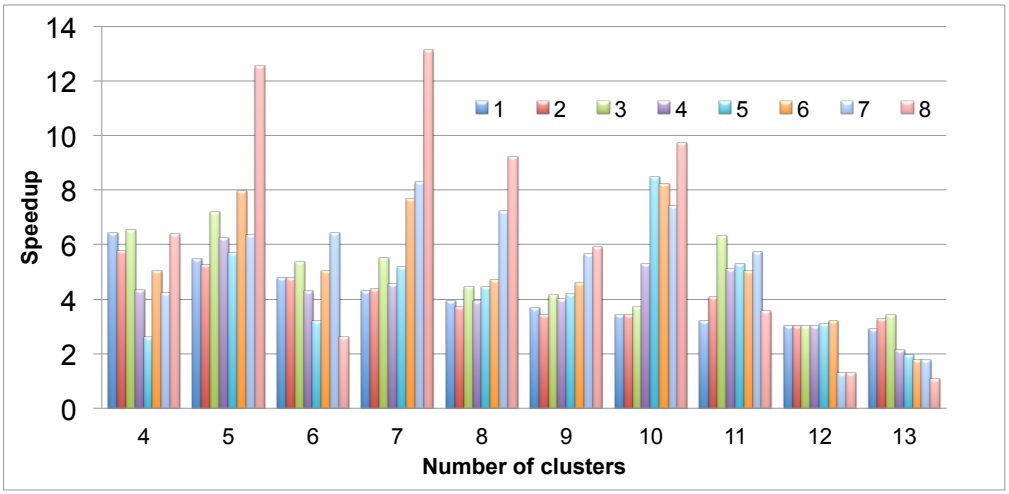


Figure 14: Speedup of relaxed versus original versions of Kmeans. Each group of bars in the X axis shows the number of clusters considered in each run of the Kmeans computation; the bars within a group represent the number of OpenMP threads used in the parallel execution. The Y axis shows the speedup of the relaxed version computed as the ratio of the original to the relaxed execution times. All the runs converged to solutions with the same quality as original.

code has a test that checks for the quality of the produced results (cluster centers). We used the same test.

Figure 14 shows the speedup computed as the ratio of the original to relaxed execution times. As with the TM version, we can see significant, up to 13x, speedups with no degradation in the quality of the results – the produced cluster centers were of the same quality as the original. The results demonstrate that the Kmeans computation has an inherent resilience under relaxed synchronization, and it can be exploited to obtain significant performance speedups.

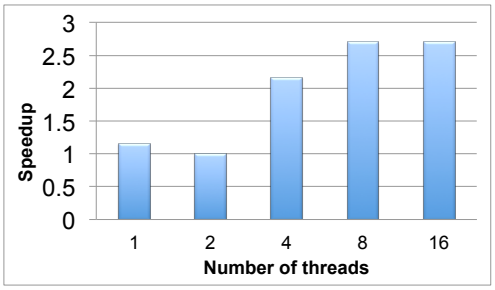


Figure 15: SSCA2 from STAMP benchmark suite. Speedup of relaxed over original computed as the ratio of the original to relaxed execution times. Quality:

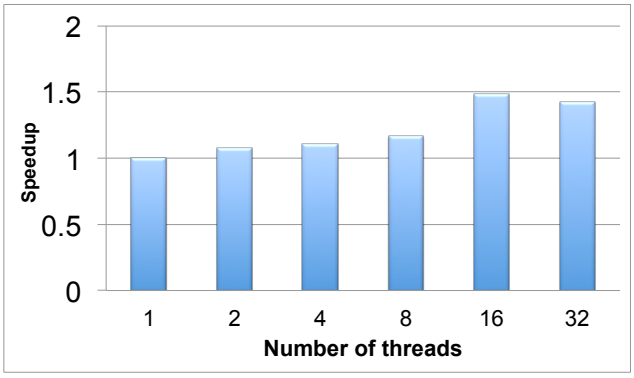


Figure 16: Labyrinth from STAMP benchmark suite. Speedup computed as the ratio of the original to relaxed execution times.

C. SSCA2

The STAMP SSCA2 benchmark is derived from the Scalable Synthetic Compact Applications 2 (SSCA2), which comprises four kernels that operate on a large, directed, weighted multi-graph. The STAMP SSCA2 benchmark performs Kernel 1, which constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. Transactions are used to protect accesses to the adjacency arrays when different threads add nodes to the graph. With a large number of graph nodes, the concurrent updates to the same adjacency list is infrequent, and hence the contention in the application is relatively low.

Figure 15 shows the speedup measured as the ratio of the original to relaxed execution times. We used the large input data set and used 1 through 16 threads.

Infrequent concurrent updates to the adjacency list makes SSCA2 an ideal candidate for synchronization relaxation. However, the concurrent update operation is relatively small, and so not much time is spent in transactions. Therefore, the benefits of relaxing synchronization will be significant only with large thread counts as seen in Figure 15. These results once again demonstrate that significant performance improvements can be obtained with no degradation in the quality of the result.

D. Labyrinth

This benchmark implements a maze-routing algorithm where the maze is represented as a uniform 3D grid. In the parallel version, each thread grabs a start and end point, which it must connect by a path of adjacent maze grid points. The calculation of the path and its addition to the global maze grid are enclosed by a single transaction. A conflict occurs when two threads pick paths that overlap. To reduce the chance of conflicts, a per-thread copy of the grid is created and used for the route calculation. Finally, when a thread wants to add a path to the global grid, it re-validates by re-reading all the grid points along the new path.

Figure 16 shows the speedup computed as the ratio of the original to the relaxed execution times. We observe modest speedups increasing with the number of threads. For the given input data set, there are 512 paths to be routed. The original version routes 512 or fewer paths on each run. The relaxed version and the original version route all 512 paths for $1, 2,$ and 4 threads. For $8, 16$ and 32 threads, the original version routes 511 paths and the relaxed version routes $509, 505,$ and 491 paths, respectively. These results illustrate the basic trade-off between performance and quality of results.

E. YADA

The YADA (Yet Another Delaunay Application) benchmark in the STAMP suite implements Ruppert's algorithm for Delaunay mesh refinement. Almost all the execution time is spent calculating the re-triangulation of skinny triangles. This benchmark has relatively long transactions and spends almost all of its execution time in transactions. During re-triangulation several triangles neighboring the skinny triangle are visited and modified, leading to large read and write sets. For this benchmark, we were able to relax only a few synchronizations which did not include memory allocation or deallocation. Consequently, the relaxation did not yield any significant performance improvement. However, in our experiments, we did not investigate the possibility of sacrificing the quality of the result. It is possible that in the context where this benchmark is used, there may be opportunities to relax the quality of the results, and hence further improvement in the speedup may be obtained using relaxed synchronization. The exploration of this dimension is part of our future work.

F. Fuzzy Kmeans

The Fuzzy K-means algorithm in the NU-MineBench allows a data object to have a degree of membership in each cluster. It is formulated in a more statistical manner compared to the original K-means formulation with a given point allowed to belong to more than one cluster with certain probability. Thus, rather than disjoint clusters, the algorithm discovers soft clusters.

Our analysis of this algorithm shows that there was a high degree of contention in updating clusters, and that this contention increases with increase in the number of threads. Consequently, relaxing the synchronization led to the algorithm failing to converge within the allocated maximum number of iterations. However, if the convergence criteria is changed, then we observe that synchronization relaxation leads to similar speedups as those observed for the original Kmeans algorithm. We omit the figures because the quality test had to be changed for this benchmark to show speedups with relaxed synchronization.

VI. Related Work

In our work, we have aimed at reducing the execution time of parallel applications using a combination of two techniques, reducing the synchronization overhead and reducing the quality of the results while keeping it acceptable. We now survey previous work in these two areas which we broadly classify as Memory Dependence Reduction techniques and Approximate Computing techniques.

A. Memory Dependence Reduction techniques

Lock-free Data Structures. Programs running on shared-memory multiprocessors typically ensure consistency of shared data by protecting critical sections with locks supported in hardware. When a thread holding the lock encounters a long-latency event, such as a page fault, other threads in the system cannot operate on the locked object and hence are unable to make forward progress. Lock-free synchronization, as described in [10], allows parallel threads to ensure consistency of a shared object while avoiding the problems of locks. A specific methodology for implementing non-blocking sharing of objects is presented in [8]. The costs of unneeded parallelism and unnecessary data copying associated with non-blocking synchronization are addressed in [1] using protocols that rely on the operating system to take corrective action whenever a thread trying to do a non-blocking update encounters a long-latency event. In contrast to this approach, our work allows lock-free updates without continually verifying the validity of concurrent updates to shared objects, relying instead on an output quality estimator to determine if the results are acceptable.

Asynchronous Iterative Methods. In [4], Baudet defines asynchronous iterative methods as those that converge even when implemented on a parallel processor without the synchronization that is normally needed between cooperating processes. He identifies sufficient conditions to guarantee the convergence of such asynchronous iterative programs.

Data Privatization. Privatization [2][11] is another important technique used to eliminate storage-based dependences in parallel loops. Variable privatization [2] is used by parallelizing compilers to improve exploitation of parallelism in the program. For example, scalar privatization removes access conflicts for a scalar variable modified in different iterations of a parallel loop by allocating a private storage for that variable in each thread. Array privatization [11], an extension of scalar privatization, creates a private copy of the array in each thread when it can be determined that such privatization is safe. Privatization eliminates the synchronization overhead incurred if atomic updates were used instead to share variables. However it comes at the cost of memory for data duplication and possibly communication overhead between threads as discussed in [7]. We attempt instead to allow updates to occur non-atomically without privatizing, risking a possible degradation in the quality of the eventual results.

Transactional Memory. Transactional memory is an alternative to lock-based synchronization. A transaction is an atomicity unit; reads and writes within a transaction are not visible to other transactions until the transaction is committed. Hardware Transactional Memory (HTM) [8] supplements multiprocessor cache-coherence protocols with a transactional cache which holds versions of updates that are made visible to other transactions only when the transaction commits. If the transaction cannot commit due to a conflict in update of values, it is aborted and needs to be re-executed. This is a source of performance overhead for transactional memory systems.

Software Transactional Memory (STM) [14] is a software method for supporting transactional programming using the Load.Linked/Store.Conditional construct common on existing machines. Optimization to support practical implementations of STM continues to be an active research area.

In this paper we show how the performance of even transactional memory systems can be improved by relaxing atomic updates that may exist in transactions. We could potentially get even greater benefit in some applications by relaxing the abort-and-restart of selected conflicting transactions at commit time.

Implicit End-of-Loop Barrier Elimination. Parallel loop implementations in programming languages like OpenMP typically force a barrier synchronization at the end of the loop. Such barriers incur performance penalty due to (a) the hardware implementation of synchronization, (b) the load imbalance between threads, and (c) the inability of the compiler to optimize across the barrier. In [9], the effectiveness of OpenMP directives is studied on NAS Parallel Benchmarks (NPB). These benchmarks are representative of computational fluid dynamics algorithms. In two of the applications, *SP* and *BT*, the outermost loops are parallelized with the *PARALLEL_DO* directive. The parallelization overhead is reduced by removing several end-of-loop synchronizations using the *OMP_END_DO_NOWAIT* directive. Although the parallelization explicitly reduces the synchronization overhead by eliminating the implicit barrier at the end of the loop, such optimizations are applied in [9] only when it is known that there are no storage-induced dependencies. Our work takes the overhead reduction one step further by relaxing the synchronization (atomic updates) within the parallel loop body, and by allowing non-atomic updates to the data.

B. Approximate Computing

Green. A system called Green is described in [3] which proposes a simple and flexible framework that allows programmers to take advantage of opportunities to trade-off quality of service (QoS) of a solution for improvements in performance and reduction in energy consumption in a systematic manner. The approximation occurs in two phases: a calibration phase, which builds a model of the QoS loss produced by the approximation, and an operational phase, which makes approximation decisions based on the QoS constraints specified by the programmer. To provide strong statistical QoS guarantees, the operational phase also includes an adaptation function that occasionally monitors the runtime behavior and changes the approximation decisions and QoS model. Although the goals of Green are primarily energy efficiency, the nature of the solution is similar to our approach where we rely on an estimate of the quality to enable focused relaxation of synchronization.

Freshener. A recent work [15] achieves scalability on massively-parallel low-latency systems by introducing non-determinism in place of synchronization and by

correcting for the resulting inconsistency. The idea is to use some fraction of the parallel threads available in the system to check on the staleness of cached values and to *freshen* possibly stale values by recomputing the values from their inputs. This allows certain type of applications that could tolerate some degree of staleness of data to respond to queries and to update requests without having to perform expensive locking of the data involved.

Reduced Resource Computation. Another recent work [13], describes mechanisms that enable computations to execute with reduced resources allowing a loss in accuracy of the results. The paper lists several general computational patterns that are amenable to resource reduction mechanisms and describes them as simple model programs. The algorithms studied in [13] are limited to sequential implementations, but in our work we have implemented parallel versions of these model computations and quantify the impact of reducing synchronization overhead on the overall reduction in the accuracy of the results. In the spirit of [13], we further map well-known problems in data-mining and clustering to simple computational models, and analyze their amenability to relaxed synchronization.

VII. Conclusion

Synchronization overhead is a major performance limiting factor in parallel applications. In this work, we set out to understand whether a program will produce acceptable results if synchronization is omitted by allowing concurrent updates to shared data when possible. We have proposed a technique which relaxes the programmer-specified synchronization and controls the degree of relaxation using a compile time parameter. Inspired by the work in [13], we have developed new model computations that lend themselves to performance improvement without serious degradation in the quality of results when synchronization is relaxed.

Our experiments with a wide class of model computations and benchmarks show that relaxing synchronization can achieve significant speedups; for example, up to 15x for the *Kmeans* benchmark and up to 70x for one of the model computations, with no degradation in the quality of the results. As part of our future work we plan to explore the dimension of sacrificing the quality of the results to gain further improvements in speedup.

The results and insights from this work suggest that it may be fruitful to investigate automatic methods to generate versions of parallel code that trade-off quality of results for improved latency specifically by relaxing synchronization. Many Web applications today are capable of providing continuous feedback about the quality of the results produced. The synchronization level in such applications can potentially be dynamically tuned to adapt to the available resources and acceptability of quality of results.

This work on relaxed synchronization is a strong indication of the potential for the general area of approximate computing, where we sacrifice the determinism and preciseness of the general computing paradigm as practiced today for improved latency, reduced energy consumption, and lower system cost, while providing results acceptably close to what would otherwise be possible.

References

[1] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, PODC '92, pages 125–134, New York, NY, USA, 1992.

[2] F. Allen, M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. A framework for determining useful parallelism. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 207–215, New York, NY, USA, 1988.

[3] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010.

[4] Gérard M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25:226–244, April 1978.

[5] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[6] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[7] Manish Gupta. On privatization of variables for data-parallel execution. In *In Proceedings of the 11th International Parallel Processing Symposium*, pages 533–541, 1997.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993.

[9] H Jin, M Frumkin, and J Yan. The openmp implementation of nas parallel benchmarks and its performance. *Technology*, (October):6, 1999.

[10] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20:806–811, November 1977.

[11] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 313–322, New York, NY, USA, 1992.

[12] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182 –188, oct. 2006.

[13] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 806–821, New York, NY, USA, 2010.

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

[15] David Ungar, Doug Kimelman, and Sam Adams. Inconsistency robustness for scalability in interactive concurrent update in-memory molap cubes. *Inconsistency Robustness Symposium*, August 2011.