Latency tolerance through parallelization of time in scientific applications

A. Srinivasan

Dept. of Computer Science, Florida State University, Tallahassee FL 32306, USA. Email: asriniva@cs.fsu.edu.

N. Chandra

Dept. of Mechanical Engineering, Florida State University, Tallahassee FL 32306, USA. Email: chandra@eng.fsu.edu.

Abstract

Emerging computing environments, such as the Grid, promise enormous raw computational power. However, effective use of such platforms is often difficult, because conventional spatial decomposition leads to fine granularity, resulting in high communication overhead. We introduce the concept of guided simulations *to parallelize along the time domain. Here, we use the fact that typically results of other simulations of closely related problems are available. In this approach, we automatically and dynamically determine a relationship between old simulations and the one being performed, and use this to parallelize along the time domain. We demonstrate the validity of this approach by applying the technique to an important application involving molecular dynamics simulation of nano-materials. In this application, spatial decomposition is not effective due to the small size of the physical system. However, time-parallelization is effective, since the granularity is much coarser. We also mention how this approach can be extended to make it inherently fault-tolerant.*

Key words: Parallel algorithm, time parallelization, molecular dynamics, temporal decomposition

1 Introduction

Motivation: Computation is becoming an increasingly important component of science. Due to increases in computational power, sufficiently realistic solutions can be obtained for an increasing class of problems of practical and commercial importance. However, for realistic solution of a wider class of problems, much greater computational power and better modeling techniques are required. An example of such a class of applications, which serves as a motivation for our work, is the development of nano-materials and products, based on a fundamental understanding of the physical phenomena that take place at the scale of a nano-meter. While the computational power required for such applications is enormous, emerging computing environments, such as the Grid, promise sufficient raw computational power. The problem here is that it is difficult to make effective use of such systems. Thus new computing paradigms are required to make effective use of such environments, and that is the focus of this paper.

The motivation for our work arises from our interest in computing nano-material properties, using Molecular Dynamics (MD) [1] simulations. Here, the state of a system is defined by the positions and velocities of a specified set of atoms. We wish to compute the state of the system at different points in time. The state at a particular time is computed from the state at the previous time step. MD based simulation, in possible combination with other continuum based techniques (e.g. Finite element methods), is expected to play a very key role in the nanotechnology revolution. However, the use of MD is severely restricted by the extremely small time increment (of the order of femto (10^{-15}) seconds) in each step of the computation, since this leads to a large number of time steps being required. The physical systems can often be small (that is, having a small number of atoms) or of moderate size, and the computational effort then arises from the large number of time steps required. Conventional spatial decomposition is not effective then, due to high communication cost, as explained below.

Conventional parallelization: We next provide a simplistic overview of parallelization, and present the limitations of conventional techniques. Conventional parallelization is by domain decomposition (which we will also refer to as spatial parallelization), where the physical system is distributed across the processors, and each processor is responsible for computing the next state for a portion of the system. However, this computation normally requires access to portions of the system residing on other processors, and so communication is required. Communication costs typically increase proportional to the surface area of the portion of the system on each processor, while the computation

[1] The specific class of problems addressed here, that of evaluating mechanical properties, should be clearly distinguished from a host of other MD problems, where physical and thermodynamic properties, such as entropy and specific heat, are evaluated. In the latter set of problems, several independent simulations starting from different states are performed, and their results averaged in some sense. This leads to a trivially parallelizable solution scheme. Such a scheme is not possible when mechanical loads are applied or when fracture can occur at any point depending on the local properties.

cost increases proportional to the volume. Since the surface to volume ratio decreases with increase in volume, the communication overhead becomes relatively small for large systems, and such parallelization is effective then. However, it is not effective when the communication latencies are high compared to the computation cost, such as in the following situations.

(1) When we want to simulate a small system for a very long time period.
(2) When massive parallelism causes the volume per processor to become very small.
(3) In computing platforms with high communication cost, as frequently encountered in distributed computing environments.

This is an important challenge to be addressed, and this paper proposes a solution strategy to this issue, using an important application in computational nanotechnology as an example.

Scalable functional decomposition: Functional decomposition, where the data is replicated, but each processor performs a different, independent, action on the data, makes the computation more coarse grained, and consequently reduces the communication overhead. It is thus attractive for the three cases mentioned above (small physical systems, massive parallelism, and high communication cost environments). It is, however, difficult to find many such independent operations that can be simultaneously performed on the system. Usually only a small number of such operations can be obtained, and this does not scale with the number of processors.

Our solution strategy is based on the notion of scalable functional decomposition. Here, we develop techniques that enable increasing number of independent operations, as the size of the parallel system increases. We use time parallelization to accomplish this.

Parallelization of time: Here, we decompose the time domain. Different processors compute the state of the physical system at different points in time. Since each of these computations is an independent set of operations on the same physical system, we have a functional decomposition that scales with the number of processors. The difficulty here is that the computation of the states is defined iteratively, with the results for time t_{i-1} being needed for computing the state at time t_i. So it is not clear how each processor can independently and simultaneously compute the state for a different point in time. The parareal idea introduced by Baffico et. al. [1] proposed an approach to deal with this problem. However, the speedup and efficiency obtained were limited, for reasons that we explain later. In this paper, we present our approach, which overcomes the bottlenecks faced by Baffico's approach. It is based on the idea of *guided simulations*, where the results of old simulations can be used to speed up the computation of future simulations.

There are two aspects to this work. (i) The general approach based on the concept of guided simulations. In this approach, a relationship is automatically and dynamically determined between a previously completed simulation and the simulation being performed, and this relationship is used to parallelize the current simulation. (ii) The specific techniques that are used to determine a relationship between different simulations. These are application dependent to a large extent.

Though the mathematical details of the technique varies from problem to problem, the conceptual framework remains unaltered, and is not restricted to the specific application being considered. So we first describe the general features of the class of relevant scientific problems in § 2. We then present the parareal approach and its limitations in § 3. We describe our approach to time parallelization, using guided simulations, in § 4. This is followed by a discussion of our application in § 5, and the description of our method for predicting the relationship between simulations in § 6, followed by experimental results in § 7. We summary our conclusions in § 8, and also mention how our technique can be extended to make it inherently fault tolerant.

2 The class of scientific applications considered

A large class of scientific application follows the general paradigm that one tracks the evolution of the state of a system with time, where the change in state is modeled through a differential equation, along with boundary/initial conditions. For example, we can describe the model generally as: $dS/dt = f(S,t)$, where S is the state, t is time, f is some function of S and t, and dS/dt gives the rate of change of the state. The physical domain where the equation is applicable is prescribed, along with the conditions at the boundary, in addition to constraints specified at $t = t_0$. This continuous problem is discretized by computing the state at discrete points in time, and the state at some time t_i is obtained as: $S_i = S_{i-1} + g(t_{i-1}, t_{i-2}, \cdots, S_{i-1}, S_{i-2}, \cdots)$, where g is some function, possibly implicitly, of previously observed states and times (typically, of the time increments $t_i - t_{i-1}$). If the initial state S_0 is known at some time t_0, then the above recurrence can be used to iteratively determine the states at other points in time. In order to achieve the required accuracy and ensure stability of the solution, the time increments need to be sufficiently small (with the exact details being very much dependent on the numerical scheme used). In this paper, we discuss a computational scheme of a more restrictive form, which is commonly encountered, where the time points are all equally spaced, and also the next state depends only on the previous state, as shown in eq. (1).

$$S_i = F(S_{i-1}). \tag{1}$$

This is done just for convenience of presentation; our approach is applicable to the more general situation too. **We use another notational convenience in eq. (1) above. The time points t_i will refer to times at which we are interested in observing the state. In order to go from t_{i-1} to t_i, we may actually need to apply several time steps of a differential equation solver.**

The time taken clearly increases with increase in the the size of the state and with increase in the number of time steps for which the computation has to be performed. When the solution is computationally intensive, parallel computation can be used to reduce the time taken. The parallelization is conventionally through spatial decomposition; however we propose a temporal parallelization here.

MD simulations apply an iterative process that results from the numerical solution of the differential equation to simulate N atoms, having 3 position and 3 momentum components each, using Newton's laws of motion. The maximum incremental time step is dictated by the vibrational frequency of the atoms, which is in the range of femto seconds. In order to reach even a microsecond of simulation, which is still well below the realistic time periods of devices, about a billion iterations are required.

3 Time parallelization through the parareal approach

The parareal approach: Time parallelization in this strategy is accomplished through an iterative process as illustrated in Fig. 1.

Let us suppose that we want to compute the state for P time steps. The computation will involve P processors with ranks in $[1, P]$. Each processor i wishes to start simulating with the state of the system at some time t_{i-1} and evolve it to the correct state at some time t_i, by performing the computation specified by F of eq. (1). If they can do this, then each processor would have performed a simulation for a different time interval, and would incur no communication overhead. The difficulty is that normally obtaining the correct start state at time t_{i-1} will require the results from processor $i-1$, thereby making the computation sequential. This is handled by an iterative procedure, with each iteration consisting of a predict-verify-correct sequence, as described below.

Iteration 0 is started by first computing an approximate solution to eq. (1) (we will later describe some methods for such prediction). We refer to this as the predictor. We define the process more formally below. If the known initial state at time t_0 is denoted by u_0, then we define $u(0, j) = u_0$ for all j, where j is the iteration number, and the 0 indicates time t_0. The predicted state at

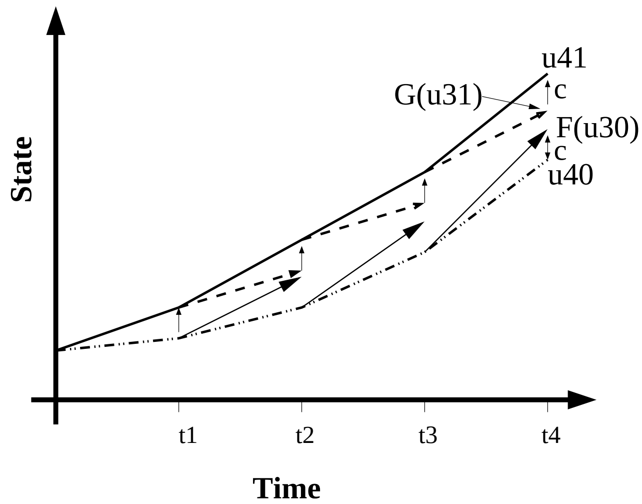

Fig. 1. Schematic of parallelization of time using the parareal approach, when the state is a single scalar. The dash-dotted line shows the prediction step in iteration 0. The solid lines with arrows denote verification step in iteration 0. The difference between the verifier and the predictor is denoted in this figure, for time t_4, by c. The predictor for iteration 1, before the corrections $c(i, 0)$ are applied, is shown by the dashed line with arrows. The actual prediction for iteration 1 is shown by the solid line without arrows.

time t_i can be obtained by the recursion $u(i, 0) = G(u(i-1, 0)), i \in [1, P]$. Here, G is an approximation to F of eq. (1), such that it can be computed fast. This phase of the computation is sequential.

The algorithm next verifies if the predicted states are accurate, in parallel. Processor i computes the state at time t_i, by starting from the predicted state at time t_{i-1} and performing an accurate computation as described by eq. (1). The resulting states are denoted by $\hat{u}(i, 0) = F(u(i-1, 0)), i \in [1, P]$. If $\hat{u}(i, 0) = u(i, 0)$ for all i, then each processor started from the correct state during its verification step, and so the final results too are accurate, and the computation can terminate. If not, then the accurate computations do not yield a correct solution, since some of their start states were incorrect.

If the predictions were not accurate, then a correction step is applied. Processor i computes the error $c(i, 0) = \hat{u}(i, 0) - u(i, 0)$. This concludes one iteration of the predict-verify-correct sequence. Note that the state $\hat{u}(1, 0)$ produced by the verifier for time t_1 is accurate, since it started from the correct, known, initial state $u(0, 0) = u_0$.

Iteration $j \geq 1$ predicts the states by first sequentially predicting the state at time t_i by the recursion $u(i, j) = G(u(i-1, j)) + c(i, j-1)$, $i \in [1, P]$. That is, the state at time t_i is predicted by applying the inaccurate predictor to the newly predicted state at time t_{i-1}, but then correcting by the error that was encountered the previous iteration. The verification and correction phase are now applied as before, to complete this iteration. If the solutions have not converged, then the computation repeats until convergence.

The computation always makes progress. For example, after iteration 1, the predicted state at time t_1 is accurate, since $u(1,1) = G(u(0,1)) + c(1,0) = G(u(0,0)) + c(1,0) = G(u(0,0)) + \hat{u}(1,0) - u(1,0) = G(u(0,0)) + F(u(0,0)) - G(u(0,0)) = F(u(0,0))$. Since $F(u(0,0))$ is the accurate solution for time t_1, the computation has made progress. This argument can easily be generalized, using induction, to show that in iteration j, the predicted state $u(i,j)$ is the exact state at time t_i for $0 \leq i \leq j$ and the verifier state $\hat{u}(i,j)$ is the exact state at time t_i for $1 \leq i \leq j+1$. Consequently, the computation progresses each iteration, finding the exact solution after P iterations in the worst case, and after one iteration in the best case. Thus, this approach always produces accurate solutions; a good predictor will improve the performance but not affect correctness, since the verifier step ensures that an incorrect computation is not accepted.

We mention a couple of schemes presented in [1] to come up with suitable predictors. One is to use a larger time step size in the solution of the differential equation. Yet another is to use a different model, which can be computed fast, but is not very accurate (for example, it might miss certain physical phenomena that the more accurate model can capture; however, if such physical phenomena occur infrequently, then the prediction will be acceptable most of the time).

Limitations of this technique: Let the number of iterations for convergence be denoted by k, and the ratio of time taken by the verifier to the time taken by the predictor by r. If we ignore communication costs, then the speedup is given by:

$$Speedup = \frac{1}{k}\frac{Pr}{P+r}, \tag{2}$$

and the efficiency by

$$E = \frac{1}{k}\frac{r}{P+r}, \tag{3}$$

One might wonder about the justification for ignoring communication costs, when one of the motivations for this approach is computing environments with high communication cost. Since time parallelization is attractive only when the time period $t_P - t_0$ is very large, for realistic numbers of processors, the time intervals $t_i - t_{i-1}$ to be simulated on each processor too will be fairly large, and so the accurate computation F too will take much time. Thus the communication costs will often be small in comparison. However, communication cost increases linearly with the number of processors (due to the sequential computation of the predictor), and may not always be negligible. Our approach will overcome this limitation.

Another factor that limits the speedup is r, the ratio of the time taken by the verifier to the time taken by the predictor. If r is very large, and convergence is attained in one iteration, then close to linear speedup is obtained. However, as P increases, the speedup is bounded above by r, even if convergence takes place in one iteration. So it is important for r to be very large in order to get good speedup and good efficiency. We argue later that it is often difficult to attain this with the techniques of [1].

The number of iterations for convergence clearly affects the efficiency significantly. Note that as the iterations proceed, even though states for many of the points in time would have already been computed correctly, the processors responsible for those points are still involved in the computation. This can be easily changed so that we do not compute for processor i, if states for $t_j, j \leq i$ have already been computed efficiently. However, the processors are still idle, and so the efficiency is still low. Our technique will overcome this problem too.

Application of the parareal technique to certain "toy" molecular dynamics simulations in [1] yielded results that were not very good, for reasons mentioned above. For example, we mentioned that a larger time step can be used in the predictor. However, the value of r attained from such a technique would be rather low in a real application. For example, in nano-materials simulations, [5] carefully chooses different time steps for different types of forces. But only a factor of 3 or 4 improvement in time is obtained by such a method. If the predictor is to be reasonably accurate, then this limits the value of r tremendously. If the predictor is not very accurate, then the number of iterations for convergence affects both, the efficiency and the speedup. A toy problem in [1] used $r = 1000$ to get a speedup of 130. However, it may be difficult to achieve this in a real simulation, for reasons mentioned above. Furthermore, the efficiency even in this toy problem was just 1.3%, even ignoring communication costs. There is one another problem with the use of a larger time step. If predictors with larger time steps are fairly accurate, then it is quite probable that an adaptive time stepping scheme would be very effective for this problem, and so a good sequential algorithm would take about the same time as the predictor, which is done sequentially anyway. Thus the speedup compared to a good sequential algorithm would not be very high. Other toy MD simulations performed by [1] use different models for prediction, which we feel is a more promising strategy. The speedups obtained are quite modest – around 8, with an efficiency of 25%, ignoring communication costs. Maday et.al. [4] use the parareal approach with a quantum control problem, and get a speedup of around 14, with the efficiency being around 1.5%.

4 Guided simulations for time parallelization

As in the parareal approach, different processors compute the state of the physical system at different points in time. (However, we divide the time period to be simulated into intervals, such that the number of time intervals simulated is much greater than the number of processors available. The choice of the size of each interval involves a trade-off between prediction efficiency and communication costs, and is explained toward the end of this section.) Our approach is to dynamically determine a relationship between the current simulation and prior simulations whose results have been recorded, enabling us to possibly predict the states at different points in time, in parallel. We then verify if the predictions are accurate, again in parallel. There is *no sequential component* in our algorithm. We give a schematic of our approach in Fig. 2.

Prediction: The most important feature of our strategy is our ability to predict the state, which serves as the starting point for each processor. The predictor should be both, accurate much of the time, and much faster than the verifier. We exploit the fact that there exists (or will come to exist) a large data base of simulation results for the same class of problems with slightly different input conditions. Further, we postulate that there will be a tangible relationship between the results of these simulations that can be effectively put to use. In this paper, we will make use of a single prior simulation, which we call the *base* simulation, obtained by solving eq. (1) for a similar system with a different input condition, as explained in § 7. We will call the simulation being performed as the *current* simulation. We dynamically determine a relationship between the base and current simulations, and this relationship is used to predict the state of the current simulation at some future point in time. The details of the prediction mechanism would be application dependent. We describe one approach in § 6 for our application. The general idea mentioned there, of using basis functions, could be applied to other problems too.

If there were a known relationship between the base and current simulations, then there is clearly no need to perform the current one, since its results can be inferred from that of the base. However, such relationships are often not known, and may only hold for a certain period in time. This implies that automatically determining such a relationship is also not easy. So, we usually do not determine the state directly from the base, but track how changes in the state of the base relate to changes in the state of the current simulation. We then use this to predict the change in the current simulation's state over a fixed interval of time. Such a scheme requires us to be able to predict only over a shorter time interval. The relationship obtained is updated on observing the error in prediction after each verification step, as we "learn" more about the relationship between the simulation being performed and the base. We shall give specific details for a class of problems in § 6.

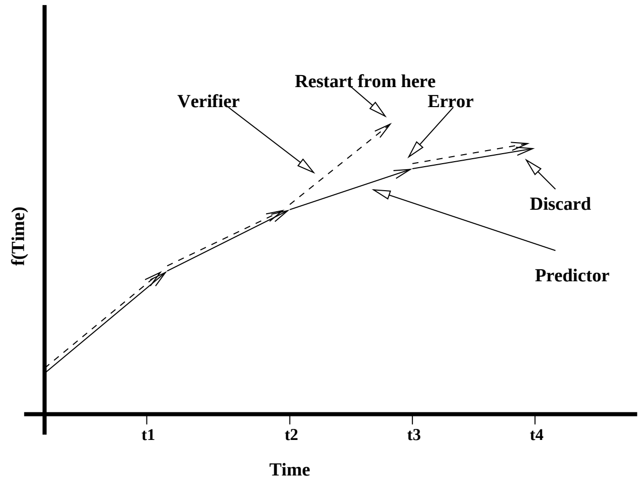

Fig. 2. Schematic of parallelization of time with 4 processors. The solid line shows the prediction, and the dashed lines the verification. Both prediction and verification are performed in parallel.

Time parallelization: Fig. 2 describes the time parallelization procedure. We will let S_i denote the state at time t_i resulting from an accurate computation that starts at the given start state S_0, T_i a predicted state at time t_i, and $\hat{S}_i$ a state at time t_i that results from an accurate computation that starts from a possibly inaccurate predicted state at time t_{i-1}. We choose the number of time points to be much larger than the number of processors. This is realistic, since we are dealing with long time simulations, and so a sufficient number of time points can be selected. In the first iteration, processor $i, i \in [1, P]$ predicts states T_{i-1} and T_i at times t_{i-1} and t_i respectively, with T_0 defined as S_0. It then performs an accurate simulation, starting from the predicted state at time t_{i-1} up to time t_i, yielding $\hat{S}_i = F(T_{i-1})$. In order to verify the accuracy of the predicted state at time t_i, processor i then compares $\hat{S}_i$ and T_i. If they differ sufficiently, then the start state for processor $i+1$ was incorrect and we say that processor $i+1$ *erred*, and its result $\hat{S}_{i+1}$ has to be discarded. In fact, the results for all processors with rank $\geq i+1$ are discarded, since they have not been verified to be correct. For example, since $\hat{S}_{i+1}$ is incorrect, we cannot be sure that T_{i+1} is correct even if $\hat{S}_{i+1}$ is sufficiently close to it. Consequently, we cannot be sure that $\hat{S}_{i+2}$ is correct, and so on. Note the following.

(1) Processor 1 can never err, since its accurate computation, $\hat{S}_1$, started from the known initial state S_0. So the computation always progresses.
(2) Difference between T_P and $\hat{S}_P$ does not lead to any computation being discarded, since T_P is not used as the initial state of any accurate computation; the comparison between $\hat{S}_P$ and T_P is performed just for the purpose of updating the predictor.
(3) All the processors must use the same predictor; otherwise verification of T_i at processor i does not imply that the prediction for T_i at processor $i+1$ was correct.

Based on the difference between T_i and $\hat{S}_i$, processor i updates its prediction scheme. The prediction scheme is an algorithm, with some parameters, which relates changes in the state of the base simulation to changes in the state of the current simulation. The relationship is reflected in the parameters to the predictor, and the parameters are updated as the predictor learns about the current relationship between changes in the base and changes in the current simulation. Given a state $\hat{S}_i$ at time t_i, and the time t_j at which a state is desired, the prediction algorithm computes a change in state between time t_i and t_j based on the prediction parameters, and adds the predicted change to $\hat{S}_i$, to predict T_j. This is shown in algorithm 4.1.

After updating their prediction parameters, the processors perform an all-reduce to determine the index k, such that the smallest ranked processor that erred has rank $k+1$. If no processor erred, then k is set to P. The prediction parameters on processor k (the last processor with validated $\hat{S}$) are broadcast to all processors, so that all processors will have the same prediction parameters. We also broadcast the state $\hat{S}_k$ computed by this processor, so that all processors can predict the change relative to the same state ($\hat{S}_k$) in the next iteration. The above procedure is then repeated, starting with the known, verified, state $\hat{S}_k$ at time t_k.

Algorithm 4.1: PREDICT(State $\hat{\mathrm{S}}_\mathrm{i}$, at time i, desired time j)

if $j == i$

then return $\hat{S}_i$

$$\textbf{else} \begin{cases} \Delta S \leftarrow \text{ a (possibly inaccurate) prediction for } \hat{S}_j - \hat{S}_i \\ \text{return } \hat{S}_i + \Delta S \end{cases}$$

Algorithm 4.2 shows the steps of guided simulations. In its description, we assume the existence of the following functions.

(1) *Predict* predicts the state of the system at a desired point in time, by first, somehow, predicting the change in state, as shown in Algorithm 4.1. The prediction mechanism is application dependent to a large extent, and we shall describe a specific predictor for our application in § 6.
(2) *AccurateComputation* performs an accurate simulation, starting at a given start state and applying the recursion defined by F in eq. (1).
(3) *UpdatePredictionParameters* Once we observe the error between the prediction and the accurate computation, we update the predictor by tuning its parameters. This clearly depends on both, the application and the predictor.

(4) *IsDifferenceTooLarge* determines whether the difference in states is too large for us to consider the prediction acceptable. This too is clearly application dependent. The above steps show that within the general paradigm of time parallelization through guided simulations, there are several application dependent algorithmic components that need to be developed.
(5) *AllReduce* and *Broadcast* can be performed using MPI. We note that the reduction to find the minimum of Next_j determines the rank of the largest indexed processor whose accurate computation has been verified as valid.

Algorithm 4.2: TimeParallelization($\mathrm{Initial\ State\ S_0, Number\ of\ processors\ P, Number\ of\ time\ steps\ m}$)

$i \leftarrow 0$

$\hat{S}_0 \leftarrow S_0$

while $i < m$

do
- **for** each processor $j \in [1, \min(P, m-i-1)]$, in parallel
  - **do**
    - $T_{i+j-1} \leftarrow$ Predict $(\hat{S}_i, i, i+j-1)$
    - $T_{i+j} \leftarrow$ Predict $(\hat{S}_i, i, i+j)$
    - $\hat{S}_{i+j} \leftarrow$ AccurateComputation(StartState $= T_{i+j-1}$, StartTime $= i+j-1$, EndTime $= i+j$)
    - UpdatePredictionParameters(CurrentParameters, $\hat{S}_{i+j}, T_{i+j}$)
    - **if** IsDifferenceTooLarge($\hat{S}_{i+j}, T_{i+j}$)
      - **then** $\mathrm{Next}_j \leftarrow j$
      - **else** $\mathrm{Next}_j \leftarrow P$
- $k \leftarrow$ AllReduce(Next, min)
- **if** $j = k$
  - **then** Broadcast $\hat{S}_{i+j}$ and prediction parameters
- **for** each processor $j \in [1, P]$
  - **do** in parallel $i \leftarrow i + k$

Numerical stability: We need to consider the numerical stability of the predictor. That is, can repeated small errors from the prediction lead to the final result being incorrect after a long period of time? We note that the new start states of each iteration of the simulation are the results of the accurate verifier computations that start from initial states that might have small errors.

Numerical schemes are normally analyzed for their numerical stability. Informally, we can describe a numerical scheme as stable if small errors resulting from the numerical approximations do not propagate or magnify with time. In our case, the errors arise from the prediction, and if they are small, the numerical scheme used in the accurate computation should not propagate it. From a physical point of view, small changes in an atom's position would cause inter-atom forces on it to push it back to the right position in our system. However, if the error is very large, then this may not happen. We expect the verifier to detect such large errors. For some of the simulations reported in this paper, we compared the results of the potential and kinetic energies of the parallel simulation against those for the exact sequential simulation, and observed that they were sufficiently close. This gives some empirical evidence for the stability of our method for this particular application.

Time complexity: Let S denote the speedup, E the efficiency, P the number of processors, m the number of time intervals simulated, n the number of iterations of the *while* loop in algorithm 4.2, τ_a the time for performing the accurate computation over one time interval, τ_p the time to perform two predictions, determine whether two states are sufficiently close, and update the predictor, τ_r the time to perform an all-reduce of an integer on P processors, and τ_b the broadcast time of the state and predictor parameters on P processors.

The sequential time is given by $m\tau_a$ and the time on P processors by $n \times (\tau_a + \tau_p + \tau_b + \tau_r)$. The speedup is thus given by:

$$S = \frac{m\tau_a}{n \times (\tau_a + \tau_p + \tau_b + \tau_r)} = \frac{m}{n} \times \frac{1}{1 + \frac{\tau_p + \tau_b + \tau_r}{\tau_a}} \tag{4}$$

In the equation above, the first fraction m/n varies between 1 and P. The former value is obtained if all the predictors fail – since the computation always progresses, only one time interval is successfully computed each iteration, in the worst case, leading to $n = m$. The latter value is obtained if all the predictors are accurate (and if P divides m). In that case, $n = m/P$. Thus, this ratio depends on the accuracy of the predictor. If we assume that $m \gg P$ and that iterations of the while loop in algorithm 4.2 lead to successful completion of αP time intervals on the average, where α is between $1/P$ and 1, then $n = m/(P\alpha) \Rightarrow m/n = \alpha P$.

We note that the second fraction in eq. (4) is close to 1 if $\tau_p + \tau_b + \tau_r$ is relatively small compared with τ_a. We have flexibility with respect to choosing τ_a for the following reason. Each time interval usually involves several steps of an accurate differential equation solver. We can make the time interval larger, implying that more steps of the differential equation solver need to be performed, and thereby increase τ_a until it becomes sufficiently large. We cannot make τ_a arbitrarily large in practice, because it is often difficult to predict accurately over long time periods, and so increasing τ_a will, at some point, start reducing α. Conversely, better predictors will enable us to increase τ_a, thereby increasing the efficiency. If we are interested in simulating for a fixed period in time, then increasing τ_a, by increasing the number of time steps of the accurate solver, will decrease m. This does not impose an important restriction to the increase, because the desired time scale is extremely large in applications involving long time scales. The times τ_r and τ_b increase with the number of processors. Under many simple models, they would increase proportional to $\log P$. However, due to their dependence on hardware, it is often more useful to determine them from empirical benchmark results. In the application for which we show results in this paper, the time for these operations is very small relative to τ_a, and the fraction $(\tau_p + \tau_b + \tau_r)/\tau_a$ varies between $\sim 10^{-4} - 10^{-2}$. Then the speedup is given by

$$S \approx \frac{m}{n} \tag{5}$$

If $m \gg P$ then we can give the speedup in terms of the empirically determined parameter α as

$$S \approx \alpha P \tag{6}$$

That is, the speedup is limited primarily by errors in the prediction. Note that efficiency in this case is given by $E \approx \alpha$, and varies between $1/P$ and 1. Note that the computation is always load balanced if $m \gg P$, since each processor performs an independent simulation of the same physical system.

The prediction time τ_p depends on the exact prediction scheme used. In this paper, the time taken for prediction is primarily that required for solving linear least squares problems. If the number of coefficients to be determined is k, and the number of atoms in the system is a, then the time taken is $O(k^2 a)$, using Householder transformations. This scales linearly with the number of atoms (size of the system), for constant k. In this paper, $k = 2$, and so the constant is small too.

Spatial parallelization: We wish to note, as an aside, that we can combine temporal and spatial parallelization; instead of one processor performing computations for the entire system at a given point in time, a group of processors will be responsible for this. For example, if the system can be parallelized efficiently through spatial decomposition on ten processors, and we have a thousand processors available, then groups of ten processors each can be assigned the task of computing for a specific time step. So time parallelization will lead to an improvement in addition to that obtained through purely spatial parallelization.

5 A nano-materials application

5.1 Specific problem

We consider a Carbon Nanotube (CNT) that has one end fixed, and the other end moved at a constant velocity u as shown in Fig. 3. Different simulations use different velocities. This is a common type of simulation to test the strength and elastic modulus of a material [2]. We wish to use as small a nanotube as possible, since our aim is to achieve as large a time scale as possible. Our application (a multiscale model in which the CNT properties, describing the nanoscale behavior of the CNT, are input to an FEM simulation) needs at least 1000 to 3000 atoms for this purpose [3]. This provides an example of an important class of applications where a small system needs to be simulated to long time scales [6].

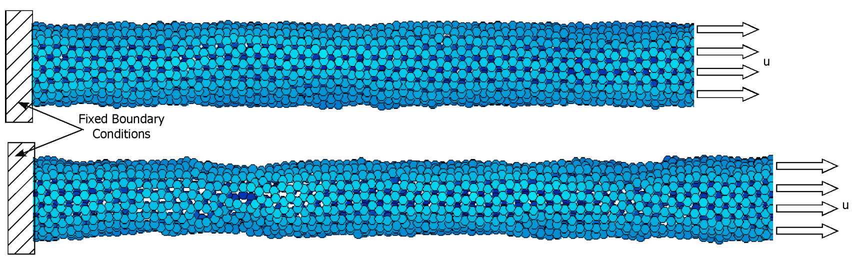

Fig. 3. Schematic of the problem. A Carbon nanotube is held fixed at one end, and pulled at the other end at a constant velocity. The higher figure shows an initial configuration, and the lower figure the configuration at a different point in time.

5.2 Equivalence of states

MD simulations bring an interesting issue – that of determining the equivalence of two dynamic states. This issue needs to be addressed since we need to determine if the predicted and accurate computations are sufficiently close. MD simulations track the vibrational motion of an atom around its mean position, as shown in Fig. 4. Even when nothing interesting is happening to the physical system, if we look at the states at two different points in time, it will be unlikely that the atoms will be in the same positions, since their vibrations cause them to behave more like a stochastic system. So we cannot expect the predicted state to have atoms in the same positions as the in accurate simulations either. Instead, we observe that the specific state observed in the MD simulation can be considered a sample from an infinite number of possible states. We need to verify if the predicted and the accurate states could be samples from the same distribution. In the parlance of MD, the two states should be samples from the same ensemble.

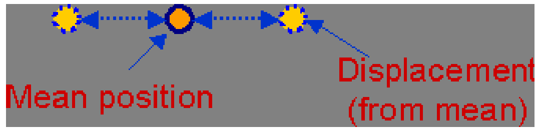

Fig. 4. Atoms vibrate around some mean position even when nothing interesting is happening to the system.

Let us consider two states having the same number of atoms, and a known mapping of atoms on one state to the atoms in the other state. This occurs in the predicted and accurate simulations, since they represent the same physical system but just assign different positions and velocities to each atom. We consider the two states to be equivalent for our purposes if the following hold.

(1) We will compute the distance between the positions of corresponding atoms in the two states. We will check if the largest distance is greater than a threshold. This threshold is determined by performing MD simulations and observing the range of movement of an atom, at the temperature of the simulation of interest, without the CNT being pulled. Based on this we choose the maximum permissible distance between two equivalent states as the largest range of movement.
(2) Similarly, we also verify if the average distance between atoms in the predicted and accurate computations is within a threshold. This threshold too, was determined as above, but considering the average distance an atom moves from its mean. It is possible for two states that are actually not identical to have the same mean and maximum deviation. So we will consider other factors.
(3) There are two types of energies that are important. The potential energy is a function of the set of positions of the atoms. However, it is quite sensitive to the positions, and two states with acceptable mean and maximum deviations can have very different potential energies. The reason for this is that the potential energy function that we use (Tersoff-Brenner potential) depends on not just the distances between atoms, but also on their orientations and dihedral angles. It is also possible for the potential energies of two states to be similar, but for the maximum deviation to be high, since one atom may not affect the potential energy much in a system with many atoms. However, a single atom moving far away is an important event to be tracked, since it may represent the material starting to break, which we wish to observe. So all the above criteria are used.
(4) In addition, the kinetic energy (or, equivalently, the temperature, which are both measured by the velocities of the atoms) is also measured. Thresholds for both the energies too were determined based on other MD simulations, and differences were required to be below the thresholds.

These threshold are shown in the graphs in § 7.

6 A predictor for the CNT application

In this predictor, we predict the change in each coordinate independently. In the description here, we normalize all the coordinates so that they are in $[0, 1]$, by letting the origin be 0 and then dividing by the length of the CNT along that coordinate direction. It is easy to change between the actual and normalized coordinates. Using the normalized coordinates is advantageous, because it enables us to use base and current simulations that use CNTs of different sizes. Let x_t represent a coordinate at time t. Using two terms of the Taylor's series, we have

$$x_{t+\Delta t} = x_t + \dot{\mathrm{x}}_{t+\Delta t}\Delta t, \tag{7}$$

where $\dot{\mathrm{x}}_{t+\Delta t}$ is the actual slope dx/dt at some point in $[t, t + \Delta t]$. We do not know the value of $\dot{\mathrm{x}}_{t+\Delta t}$, but will try to predict it.

We will consider a finite set of basis functions, $\phi_0, \phi_1, ..., \phi_k$, which are functions of the coordinates of the atom positions, and express $\dot{\mathrm{x}}$ in terms of it. For example, we can take a polynomial basis $1, x, x^2$. These basis functions should ideally be chosen so that they represent the types of changes that can occur under physical phenomena that the CNT might experience.

Let $\dot{\mathrm{x}}_{t+\Delta t} \approx \sum_i a_{i,t+\Delta t}\phi_i(x_t)$. Once we have performed an accurate simulation for time t, we know the actual $\dot{\mathrm{x}}_t$ for each atom, and can perform a least squares fit to determine the coefficients $a_{i,t}$. We can express changes in the base similarly, and determine its coefficients, say $b_{i,t}$. The values of $b_{i,t}$ are available for the entire duration of the base simulation. The coefficients $a_{i,t}$, on the other hand, are computed and are only available until the time step simulated so far. If the base simulation and the current simulation are almost identical, then we can approximate $a_{i,t+\Delta t}$ by $b_{i,t+\Delta t}$. However, the simulations will typically differ, and so we wish to correct by adding the the difference for the two simulations $R_{t+\Delta t} = a_{i,t+\Delta t} - b_{i,t+\Delta t}$, which is unknown. As a first approximation, we can assume that $R_{t+\Delta t} = a_{i,t} - b_{i,t}$ to yield the approximation $a_{i,t+\Delta t} \approx b_{i,t+\Delta t} + a_{i,t} - b_{i,t}$. On one hand, using the latest $a_{i,t}$ available might give the best estimate. On the other hand, random fluctuations in the MD simulation lead to somewhat poor results if we depend on only evaluation at one point in time. So we set $R_{t+\Delta t} = (1 - \beta)R_t + \beta(a_{i,t} - b_{i,t})$, where β is the weight assigned to the latest value. We note that the point in time t mentioned here is the latest point at which the actual coefficients are available, and need not coincide with the t in eq. (7). The term R_t represents the relationship between the base and current simulations, and updating it at each time step represents a simple form of learning.

Since $a_{i,0}$ and $b_{i,0}$ are unknown, we need to chose a suitable method of starting this process. We assume a linear increase with time, in the values of the coordinates of atoms in the direction in which the CNT is pulled, with the constant of proportionality being a function of its normalized coordinates for our experiments. This is sufficient for a moderate time interval, but fails after that. Of course, the prediction-verification process detects the errors and corrects for it.

The velocity distribution of the atoms was predicted to be the distribution at the previous point in time for the current simulation. Since the numerical simulations were carried out at constant temperature, this was sufficient.

7 Experimental results

Experimental parameters: Both the base and the current simulation pull one end of a Carbon nanotube containing 1000 atoms at constant velocities, but with different velocities for the base and the current simulations. The predictor, of course, did not make use of the fact the the same number of atoms were simulated. The Tersoff-Brenner potential is used in the MD simulations. Use of this potential is more computationally intensive than many other potentials, but is known for its accuracy in Carbon nanotube calculations. One end of the tube is kept fixed by forcing around 100 atoms at that end to remain stationary. The tube is pulled at a fixed rate in the z direction by forcing around 100 atoms at the other end to increase their z coordinate values at a rate of u Angstroms per time step, where u is 0.00005 for the base simulation, 0.0000625 for the simulation being performed, and the time step is 0.5 femto seconds. The remaining around 800 atoms are simulated using MD, with the end atoms providing the boundary conditions. Note that with just 800 atoms, we cannot achieve much speedup through spatial parallelization, since the granularity will become very fine. Most published results [7] on codes using the Brenner potential require around 1000 atoms per processor for good efficiency. Our spatially parallelized code is effective for less than 500 atoms per processor, but even then does not yield good efficiency on more than $2-3$ processors, for a system of this size.

We first performed an accurate MD simulation for the base case (since it is needed for the prediction) at a temperature of around $10K$, and recorded the output of every 1000 time steps, over $200,000$ time steps. The new simulation was performed at a temperature of $300K$, and thus our results demonstrate that we can predict a relationship between simulations performed under different conditions. The predictor used $\beta = 0.5$, and the basis functions used were the first two terms of the polynomial basis, 1 and x. The results are not very sensitive to the value of β, as long as β is not close to 0 or 1.

A parallel run was performed on the following machines.

(1) A 32-node Ethernet switched cluster. The nodes of the cluster are Dell Dimension 2400 with a 2.8GHz P4 processor each, 128MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to Dell PowerEdge 2224 and Dell PowerEdge 2324 100Mbps Ethernet switches. The topology is a star with four switches, with each switch having around 8 processors attached to it.
(2) A 168 processor IBM eServer pSeries 690 system. It is made up of 42 nodes, each consisting of four IBM Power3 based processors with a clock speed of 375 MHz, and with 2 GB memory per node, and 36 GB of disk space. This is a shared memory machine, and its MPI communication mode was set to make use of the shared memory for improved performance.

The sequential code takes around $16,000$ msec for one time interval (1000 time steps) on the Linux cluster, and around $55,000$ msec on the IBM machine. In contrast, the broadcast (which is the dominant communication cost) requires around 460 msec on the cluster when 32 nodes are used (and the order of 100 msec for smaller numbers of processors), and around 5 msec on the IBM SP3 even with 50 processors. The predictor takes the order of 1 msec. Thus the parallelization overheads are small.

Results: We first show in Fig. 5 the difference between the predictor and the verifier for each of the four quantities that define the equivalence of two states. This simulation is performed on 32 processors. The dotted lines indicate the error thresholds, which are determined based on the natural fluctuations in MD simulations under similar situations. Each solid line represents a different iteration of the while loop in algorithm 4.2. All the results are below the respective thresholds, indicating that errors are sufficiently small. We can see that the errors keep decreasing over time, indicating that the predictor learns better about the relationship between the two simulations. Note the following:

(1) The potential energy threshold permits a fluctuation of around 0.7%.
(2) The desired temperature of the simulation is $300K$.
(3) The 200 time intervals reported here represent a total of $200,000$ steps of MD simulations, since each time interval involves 1000 MD steps.
(4) The errors decrease rapidly after the first iteration, since the predictor then "knows" something about the relationship between the base and the

current simulation.

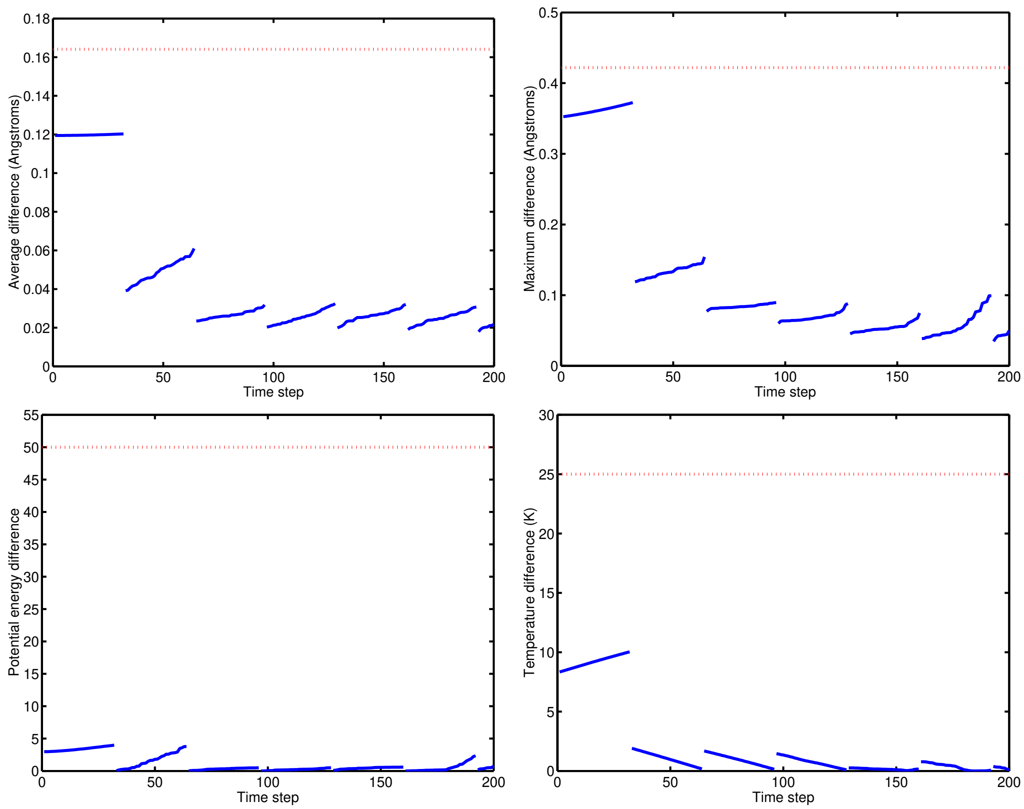

Fig. 5. Error comparing the predicted state with the accurate verifier computation with $P = 32$. The dotted lines indicate the error thresholds. Each solid line gives the error for a different iteration of the while loop in algorithm 4.2.

In Fig. 6, we show the speedups on the two platforms where the code was tested. The efficiency is greater than 90% for up to 50 processors. We can observe that the speedups are similar on the IBM machine and on the Linux cluster, though communication is much faster on the IBM machine than on the cluster. For the numbers of processors shown in the figures, the prediction never errs (that is, the error is below the threshold, and so we do not need to discard any computation). The loss in efficiency is not due to the communication or prediction overheads either, since they are small, as explained earlier. All the processors read from and write to files present on a common machine. File I/O is not accounted for in our model, and it appears to be the cause for efficiency lower than expected. Non-blocking I/O is a potential solution to this problem, though the efficiency is high even without it.

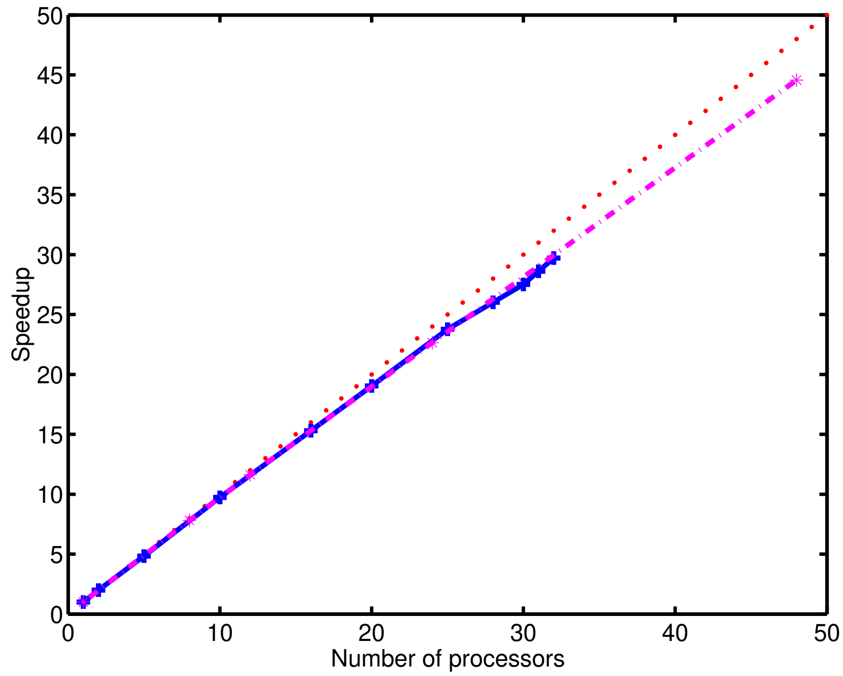

Fig. 6. Speedup using time parallelization. The dotted line is the ideal perfect speedup. The solid line is the speedup on the Linux cluster, with the data points marked by '+' signs. The dash-dotted line is the speedup on the IBM machine, with the data points marked by '*' signs.

8 Conclusions and future work

We have proposed a strategy which can enable high latency environments to be effective for problems where a typical spatial decomposition will lead to too fine a granularity. We have demonstrated its effectiveness on an important and real application of great technological significance.

This method can be combined with spatial parallelization for even greater effectiveness. Of course, further work needs to be performed to extend the applications for which this technique can be used.

We note that the time parallelization approach can be extended to produce algorithms that are inherently fault tolerant, without the need for check-pointing. Instead of an error being due just to poor prediction, it will also be due to failure of processes or processors. However, in the case of these types of failures, we will not need to discard the result of subsequent time steps, since they have not really been invalidated. We just need to “fill in” the missing points in time.

Acknowledgements

This work was funded by NSF grant # CMS-0403746. We also wish to thank Sri S.S. Baba for help with debugging the code, among other things.

References

[1] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zerah. Parallel-in-time molecular-dynamics simulations. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 66:57701–57704, 2002.

[2] N. Chandra, S. Namilae, and C. Shet. Local elastic properties of carbon nanotubes in the presence of stone-wales defects. *Physical Review B*, 69, 2004.

[3] N. Chandra, S. Namilae, and A. Srinivasan. Linking atomistic and continuum mechanics using multi-scale models. In *Proceedings of the 8th International Conference on Numerical Methods in Industrial Forming Processes, Numiform 2004*, 2004.

[4] Y. Maday and G. Turinici. Parallel in time algorithms for quantum control: parareal time discretization scheme. *International Journal of Quantum Chemistry*, 93:223–238, 2003.

[5] A. Nakano, P. Vashishta, and R. K. Kalia. Parallel multiple-time-step molecular dynamics with three-body interaction. *Comput. Phys. Commun.*, 77:303–312, 1993.

[6] S. Namilae, N. Chandra, and C. Shet. Mechanical behavior of functionalized nanotubes. *Chemical Physical Letters*, 387:247–252, 2004.

[7] D. Srivastava and S. T. Bernard. Molecular dynamics simulation of large-scale carbon nanotubes on a shared-memory architecture. In *Proceedings of the IEEE/ACM SC1997 Conference*. IEEE Computer Society, 1997.