PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Efficient large-scale replica-exchange simulations on production infrastructure

Abhinav Thota, André Luckow and Shantenu Jha

Phil. Trans. R. Soc. A 2011 **369**, 3318-3335
doi: 10.1098/rsta.2011.0151

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

Phil. Trans. R. Soc. A (2011) **369**, 3318–3335
doi:10.1098/rsta.2011.0151

Efficient large-scale replica-exchange simulations on production infrastructure

By Abhinav Thota[1,2], André Luckow[1] and Shantenu Jha[1,3,*]

[1]*Centre for Computation and Technology, and* [2]*Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA*
[3]*Rutgers, State University of New Jersey, NJ, USA*

Replica-exchange (RE) algorithms are used to understand physical phenomena—ranging from protein folding dynamics to binding affinity calculations. They represent a class of algorithms that involve a large number of loosely coupled ensembles, and are thus amenable to using distributed resources. We develop a framework for RE that supports different replica pairing (synchronous versus asynchronous) and exchange coordination mechanisms (centralized versus decentralized) and which can use a range of production cyberinfrastructures concurrently. We characterize the performance of both RE algorithms at an unprecedented number of cores employed—the number of replicas and the typical number of cores per replica—on the production distributed infrastructure. We find that the asynchronous algorithms outperform the synchronous algorithms, even though details of the specific implementations are important determinants of performance.

Keywords: replica exchange; SAGA; large-scale distributed cyberinfrastructure

1. Introduction

Replica-exchange (RE) [1,2] methods have been used to understand physical phenomena—ranging from protein folding dynamics to binding affinity calculations. The design and development of most RE runtime implementations [3] are influenced by the specific infrastructure they are developed upon and constrained by the programming systems used. Interoperability across infrastructure and extensible functionality are typically not first-class considerations. Breaking this coupling between the development and runtime environments, on the one hand, and the runtime environment and underlying infrastructure, on the other, is an important design objective of distributed applications—either logically distributed or physically distributed. It enables applications to be flexible (across infrastructure), extensible (to new methods of communication and coordination) and scalable.

Application formulations that are flexible are better suited to using the diverse range of traditional and hybrid infrastructure (e.g. grid–cloud and heterogeneous resources) in a scalable manner. Along with application formulations that

*Author for correspondence (sjha@cct.lsu.edu).

One contribution of 12 to a Theme Issue ‘e-Science: novel research, new science and enduring impact’.

facilitate the scalable and flexible use of a range of infrastructures, it is imperative to have the correct runtime abstractions that support flexible deployment of these applications. This work brings together advances in application algorithms, along with sophisticated runtime environments to support these algorithms. Specifically, in order to support flexible and scalable formulations of the RE class of algorithms, we develop an RE framework that supports multiple formulations, is extensible to a broad range of infrastructures and, as we shall show, scales up and scales out.

The RE framework uses a flexible pilot-job implementation—Simple API for Grid Applications (SAGA) BigJob [4]—to support the efficient execution of ensembles. The pilot-job provides a container for a number of sub-jobs (replicas), which can then be directly and concurrently executed via the pilot-job, and thus circumventing the need for replicas to individually wait for resources to become available. The RE framework supports scalable implementation of RE that can use a range of infrastructures concurrently and that supports different replica coordination mechanisms, different exchange coordination mechanisms (synchronous versus asynchronous) and thereby different variants of the RE algorithm.

The paper is organized as follows. Section 2 sketches out the two different RE algorithms that are investigated; we also present an approximate mathematical model for the different algorithms. Section 3 outlines the architecture of the RE framework—the SAGA BigJob (how it supports the dynamic execution of multiple replicas) and other important elements that make the framework flexible and extensible. In §4, we present our implementation of the RE algorithms and understand the primary determinants of performance and relate it to the mathematical model in §2. In §5, we describe the experiments performed to assess and understand performance when scaling up (on a single machine) as the number of replicas increases. We compare and analyse the performance of the different RE formulations (synchronous and asynchronous) when scaled up to 256 replicas as well as when scaled out to use more than one machine. The physical system that we use as the benchmark is the hepatitis C virus that was examined in the study of Luckow *et al.* [5]. Section 6 concludes the paper and discusses future work.

2. Replica-exchange algorithms

The RE class of algorithms involves the concurrent execution of *replicas*, which are defined as instances of essentially similar simulations but with minor differences, such as the defining temperature of the replica. These replicas are loosely coupled, in that there are infrequent exchanges between pairs of replicas. In addition to the frequency of communication between the replicas being low (relative to that within a single replica), the amount of information/data exchanged between replicas is small (a couple of bytes) compared with the simulation's operating dataset size.

(*a*) *Mathematical model*

In this subsection, we develop a mathematical model that captures the primary components that make up the total runtime of an RE experiment. In an ideal scenario, the total time to completion of an experiment would be equal to

the concurrent runtime of the ensemble of replicas and there would be no overhead associated with the coordination of the replicas. We also assume that the resources, network and other components are homogeneous. If an ensemble contains N_R replicas, and the total number of pairwise exchanges is defined to be N_X and the runtime of a replica to complete a defined number of time steps is defined to be T_{MD}, then the total time to completion of an experiment T would be

$$T = \frac{1}{p} \times \left(T_{MD} \times \frac{N_X}{N_R/2} \right), \tag{2.1}$$

where p is defined as the probability of a successful exchange. The decision to accept an exchange or not is made using the Metropolis scheme [6], which is a well-known way of accepting a proposed change of state, even when energetically not favourable.

$N_R/2$ is the number of independent exchange events that can occur concurrently for N_R replicas. $N_X/(N_R/2)$ is the number of ensemble runs needed to complete N_X exchanges.

For example, if N_R is 4 and N_X is 16, two exchanges are possible after an ensemble run and eight such runs are required to complete 16 exchanges. After the exchange, the replicas are restarted with the new temperatures. However, any RE production run will entail some overhead of job submission, termination, coordinating the replicas and exchanges, etc. Thus, the time to complete an RE simulation of N_X exchanges successfully is

$$T = \frac{1}{p} \times \left[\left(T_{MD} \times \frac{N_X}{N_R/2} \right) + (T_{EX} + T_W) \times \frac{N_X}{\eta} \right], \tag{2.2}$$

where T_{EX} is the time to perform a *pairwise* exchange. It includes the following components: (i) the time to find a partner (T_{find}), (ii) the time to exchange/write/transfer files (T_{file}), and (iii) the time to manage state updates (e.g. in a central database) and conduct book-keeping operations associated with replica pairing/exchanging (T_{state}). T_{state} may arise for different reasons, which may be related to implementation of the different RE algorithms. In summary, $T_{EX} = T_{find} + T_{file} + T_{state}$. The last component T_W is the waiting time spent by a replica waiting to synchronize with other replicas that are not ready, e.g. perhaps still running. η describes the number of concurrent exchanges taking place. At maximum η is $N_R/2$; it can however be lower if exchanges cannot be conducted concurrently. It can be seen from equation (2.2) that, as T_{MD} increases, the cost of coordination becomes less relevant. For simplicity, yet without loss of generality, we use a fixed value of p ($=1$) in this work.

(*b*) *Synchronous replica exchange*

Traditionally, RE algorithms have been implemented such that the exchanges have been synchronous. If the number of replicas is N_R, a constant number ($N_R/2$) of *fixed* replica pairs is generated. When *all* the replicas in the ensemble reach a defined state (e.g. the molecular dynamics (MD) simulation completes a defined number of steps), an exchange of temperatures between the fixed and the pre-determined paired replicas is attempted using the Metropolis scheme. If the exchange attempt is successful, parameters such as the temperature are swapped.

For the synchronous RE formulation, all replicas must reach a pre-determined state before exchanges are performed. T_{W} is the time waiting for all the replicas in the ensemble to reach this state. In contrast to parameter sweep applications in which coordination commonly only occurs at the beginning and the end—a pattern commonly referred to as scatter–gather—RE requires periodic coordination. In the synchronous RE formulation, coordination occurs at defined intervals.

A major limitation of this model is that the replicas are paired in fixed groups and thus exchanges take place between pre-determined pairs of replicas. As a consequence of pairs being determined before an exchange, although T_{find} is 0, this limits the number of possible exchange partners that are available for a given replica. This inhibits exchanges between replicas with non-nearest temperatures, and ultimately reduces the possibility of crosswalks—where a crosswalk is said to occur when a replica originally with a low temperature reaches the upper temperature range and then returns to the lower temperature range [7].

In addition to limitations in modelling the physics, rigid replica pairing is efficient only in homogeneous environments; for heterogeneous environments and systems, where resource availability and performance fluctuate, the need for synchronization leads to slow down and inefficiencies. We show how these limitations are overcome in the asynchronous (exchange) formulations of RE.

(*c*) *Asynchronous replica exchange*

In asynchronous RE algorithms [7,8], a replica does not have to wait for *all* other replicas to reach a pre-determined state. An exchange can occur whenever a replica reaches a pre-determined state. The replica attempts an exchange with another suitable replica in the ensemble.

In principle, the issue of *static* versus *dynamic* pairing is independent of synchronous or asynchronous exchanges, in that one could have synchronized exchanges but with a different replica every time; equivalently, we could have fixed pairs, but no global synchronization, i.e. asynchronous exchange between fixed pairs. However, in this work, we will equate synchronous exchange with static pairing, and asynchronous exchanges with dynamic pairing.

Thus, for the asynchronous algorithms, as each replica on completing a run has to find a *new* partner, $T_{\mathrm{find}} \neq 0$; however, T_{W} is 0 because there is no synchronization involved. In other words, a reduction in *synchronization* (wait) times comes at the cost of increased *coordination* (replica pairing) costs. The specific value of the term T_{EX} (equation (2.2)) differs from the synchronous formulation.

3. Replica-exchange framework

An important motivation for and contribution of this work is the design and implementation of a framework that provides the capability to support different RE algorithms and exchange mechanisms. The framework is independent of the underlying infrastructure and thus supports the use of multiple heterogeneous infrastructures—which are typically made available to the end-users. It is useful to highlight that we differ from other RE implementations (e.g. [7]) in that we use *production-grade* national and regional cyberinfrastructures, such as the

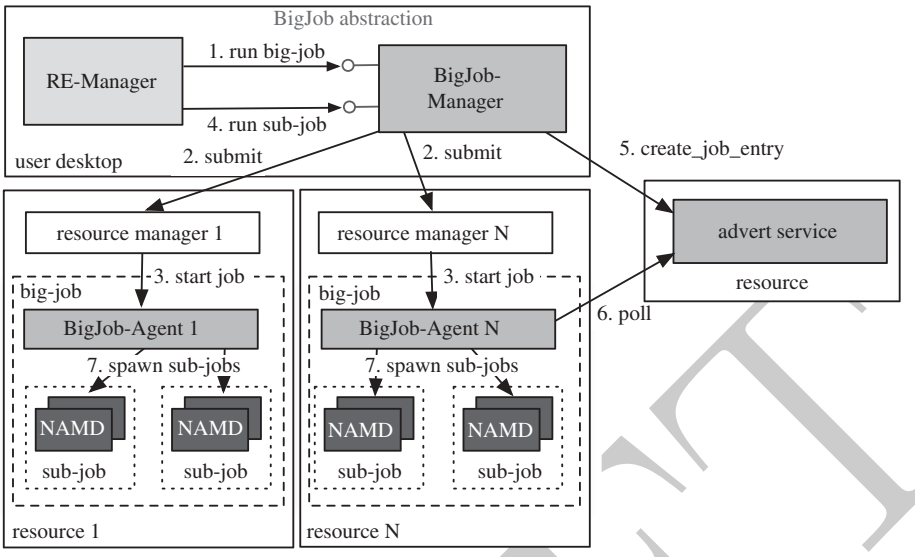

Figure 1. The RE framework. The framework consists of the RE-Manager and the SAGA BigJob framework. BigJob is used to efficiently execute replica sub-jobs. The RE-Manager manages replica sub-jobs and performs REs. Light grey, RE-Manager; dark grey, SAGA BigJob; black, application.

US TeraGrid and LONI (http://www.loni.org/), using general purpose tooling and standard capabilities that are available on these production infrastructures. Additionally, our framework *natively* supports individual replicas that are MPI jobs.

In this section, we outline the architecture, implementation and the basic performance of the RE framework when used to implement the different RE algorithms (synchronous and asynchronous) and exchange mechanisms (centralized and decentralized). The RE framework consists of the RE-Manager and the SAGA BigJob framework, which is used to efficiently execute the ensemble jobs. The BigJob framework is common for both synchronous and asynchronous RE. The RE-Manager's functionality varies in the different implementations—centralized or decentralized.

(*a*) *SAGA BigJob: a pilot-job framework*

Figure 1 shows the architecture of the RE framework—which comprises the RE-Manager and SAGA BigJob framework.

SAGA (http://saga.cct.lsu.edu) is an application programming interface (API) for the basic functionality required to build distributed applications, tools and frameworks. SAGA was designed to be independent of the details of the underlying infrastructure. We have previously demonstrated the usage of the SAGA-based pilot-job framework [4]—called the big-job—to run RE simulations across multiple, heterogeneous, distributed grid and cloud infrastructures [5].

The SAGA BigJob framework consists of three components: (i) the BigJob-Manager, (ii) the BigJob-Agent, and (iii) the advert service which is a central key/value store used for communication between the BigJob-Manager and the BigJob-Agent. The various tasks that are carried out using the SAGA APIs

include file staging, job spawning and conducting the exchange attempts. Here we use the SAGA BigJob framework to efficiently request and manage computational resources for multiple replicas and it enables the use of a range of infrastructures. For example, we can submit multiple big-jobs on multiple resources and manage them from one location.

The RE-Manager requests a defined number of big-jobs from the BigJob-Manager; for each big-job, a regular batch job is submitted to the resource manager (steps 1–3 in figure 1). When the BigJob becomes active, the BigJob-Manager can start to process sub-jobs. For each new sub-job, an advert entry storing the description of the sub-job is created by the BigJob-Manager (steps 4 and 5). The BigJob-Agent periodically polls for new jobs (step 6). If a new job is found and resources are available, the BigJob-Agent runs the job (step 7). If not, the job is queued. It is possible that there are more sub-jobs than the big-job can accommodate. Further, the agent continuously monitors the running sub-jobs and updates the sub-job states in the advert service. Once a sub-job finishes running, the compute resources are freed and marked as available.

(*b*) *Replica-exchange manager*

The RE-Manager is the master process which in addition to controlling the different components—SAGA BigJob framework, the individual replicas, etc.—also defines and implements the coordination/exchange mechanism employed. The actual tasks that the RE-Manager performs depends not only on the RE algorithm employed but also on which exchange mechanism is being supported. The RE-Manager supports two different replica management and exchange mechanisms: centralized and decentralized. In the centralized case, the RE-Manager manages all the replicas and performs the exchanges, while in the decentralized case a *replica agent* manages each replica individually as well as making and performing exchange decisions.

As we will discuss in §4, the synchronous and asynchronous RE algorithms were implemented using the centralized RE-Manager; the decentralized RE-Manager was used to support only the asynchronous RE algorithm.

(i) *Centralized replica-exchange manager*

The control flow of a centralized RE coordination mechanism is shown in figure 2*a*. A replica can be in one of these three states: (i) `new` (submitted but not started), (ii) `running`, and (iii) `done`. Once the big-job(s) is active and replicas are `running`, the RE-Manager constantly queries the BigJob-Manager for the latest replica states. When the RE-Manager finds a replica that has finished running, it collects the energy and temperature of that replica by reading the output file. Once *all* the replicas have finished running, the RE-Manager performs the exchanges by swapping temperatures and writing new configuration files. The new configuration files are staged to the appropriate location. The RE-Manager then submits the replicas for restarting, and the BigJob-Manager restarts them. The RE-Manager keeps count of the successful exchanges, until the required number of exchanges are done.

We implemented both the synchronous and asynchronous RE using the centralized version of the framework. Even though both use the same framework, the implementation of the asynchronous RE-Manager is different from the

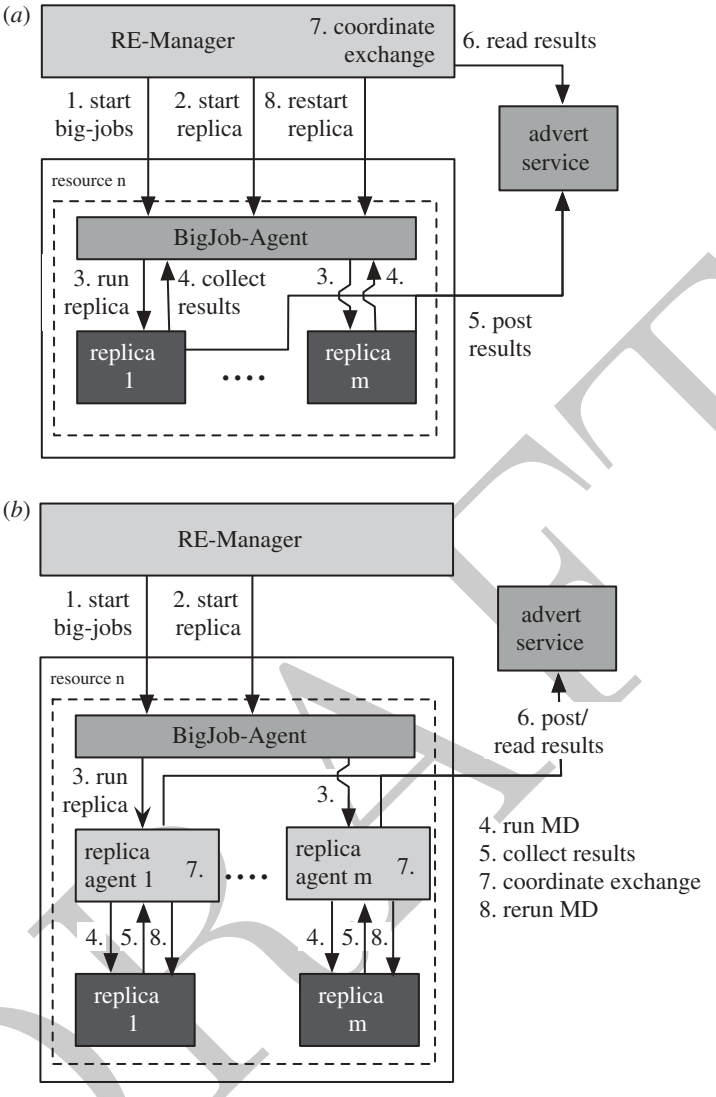

Figure 2. (*a*) Centralized versus (*b*) decentralized RE coordination. Both synchronous and asynchronous RE are implemented using the centralized coordination style. Asynchronous RE is also implemented using the decentralized style, where the master initially sets up all the replicas. The RE coordination is done peer-to-peer via the Advert Service. Light grey, RE-Manager/Agent; dark grey, BigJob; black, application.

synchronous RE-Manager. The asynchronous RE-Manager, for example, is not required to wait for *all* replicas to finish running before performing *all* exchanges. Whenever the asynchronous RE-Manager finds a replica that has finished running, it tries to find a partner to make an exchange. To find a partner, the RE-Manager goes over the list of all the replicas in the ensemble. If it finds a replica available it attempts the exchange. If a replica is not found available, the RE-Manager queries the BigJob-Manager for the latest replica states and

updates its local list. It then loops over the list to find a replica that has finished running and a partner to exchange with that replica. If successful, the replicas are submitted to be restarted.

(ii) *Decentralized replica-exchange manager and the replica agent*

Additional states are needed to implement and maintain decentralized coordination/control. Thus, the states that a replica can be in the decentralized RE coordination mechanisms are different from those in the centralized mechanisms. In the centralized implementation, there is no `complete` state and in the decentralized implementation there is no `new` state. There is no `new` state in the decentralized implementation because the replica agent starts running only after the processors become available and the replica is started and put directly in the `running` state.

A replica can be in one of the these four states: in addition to states (i) `running`, (ii) `done`, which were found in the centralized case, there are also (iii) `pending` (for an exchange), and (iv) `complete` (exchange has been completed) states. When a replica reaches a pre-determined state, it (the replica agent acts as the proxy for the replica) transitions from `running` to `done`. It initiates the search for a partner, and scans the list of replicas randomly, so as to avoid contention if multiple replicas have initiated a search for a partner.

If a replica finds a potential exchange partner, it must re-verify that both states are still in the `done` state. This step is necessary to avoid concurrency issues that arise if multiple replicas attempt to exchange with the same partner. If both replicas are still in `done` state, the exchange can proceed and their states are set to `pending`. If the state of any replica has already been changed (to `pending`), the exchange attempt is aborted.

After the exchange is performed (the temperature of both replicas in the advert server has been changed), their states are set as `complete`. Here, the state `complete` is a marker which tells the replica agent that the exchange has been made, the configuration files are in place and to restart the replica. The associated replica agents write new configuration files with updated temperatures, and restart their replicas. The replica agent that initiated the exchange increments the exchange count. The RE-Manager constantly queries the advert server for the latest exchange count and, when all exchanges have been made, it stops the experiment.

In the decentralized implementation, in order to conduct the exchanges, the RE-Manager launches multiple replica agents (in lieu of replicas directly). Replica agents then take control of replica start/restart and exchange attempts. Figure 2*b* shows the control flow in the decentralized RE coordination mechanism. The replica agents, upon launch, run the replicas; a list of nodes that are used to carry out the MD run is passed to the replica agent as an argument at start-up. The replica agent constantly monitors the replica, and, when the replica finishes, it updates the advert server with the current state of that replica. It also reads the temperature and energy from the output files, and posts the values to the advert server. The RE-Manager is primarily responsible for keeping track and count of the number of exchanges performed; when the desired number of exchanges are done, the RE-Manager ends the experiment.

It should be noted that, even in the decentralized implementation, the advert service is still centralized. Only the replica management and exchange coordination are decentralized. We did not observe any increase in latency with respect to the increasing number of connections when accessing the advert service. But the communication times do vary depending on the physical distribution of the advert service. Also, maintaining data consistency is not an issue because there is only one advert server. Two write actions cannot be carried out simultaneously on a single value. It should also be noted that the exchanges are non-deterministic in nature, first because the replicas available for exchange are found in a random order and, second, because, in principle, the Metropolis scheme [6] is used to match the replicas.

4. Executing multiple replica-exchange algorithms using the replica-exchange framework: implementation and performance

In §3, we presented the basic components of the RE framework and discussed the control flow for the three implementations of the two algorithms using different exchange coordination mechanisms. In this section, we provide further details of the working of the RE framework, as well as a basic characterization of the performance of the RE framework for the three formulations.

For the basic characterization, the following experimental configuration is used. (i) Infrastructure: our experiments are performed on LONI and the TeraGrid shared resource *QueenBee* (*QB*). A highly scalable, parallel MD code—NAMD [9]—is used to perform the simulations for each replica (although it is important to mention that any other MD or Monte Carlo code could be used just as simply and effectively with the RE framework). (ii) RE configuration: the total number of replicas (N_{R}) in the ensemble is 32 and the total number of pairwise exchanges (N_{X}) is 128. As the ensemble of replicas is run concurrently, 16 pairwise exchanges are possible after each concurrent run. Thus, each replica on average is restarted seven times. Each replica is configured to run 500 time steps and is allocated 16 processors. One big-job of size 512 processors is requested. On average each 500 time-step run takes 71 s. For all implementations, in the event of a successful exchange, jobs are restarted [5] with new temperature values. In the case of an unsuccessful exchange, jobs are restarted without exchanging the configuration. (iii) The physical system that we use as the benchmark is the hepatitis C virus that was examined in Luckow *et al.* [5]. It is to be noted that, in actual biological science simulations, the number of time steps between exchanges is often higher than 500. Thus, the amount of computation performed between exchanges is typically higher, thus lowering the cost and relevance of performing the exchange.

Each production run was repeated multiple times (≈10) and the start time of each run is measured only after the big-job becomes active. As we are interested in understanding the scale-up and scale-out properties of synchronous and asynchronous RE, we do not consider queue wait times. As explained in §2*a*, the relative performance of RE implementations is determined by the waiting time T_{W} and the time for conducting the exchange T_{EX}. In the following sections, we analyse the average values for T_{W} and T_{EX} for each replica pair. Table 1 summarizes the results. We will discuss the different RE implementations

Table 1. Average values of terms in equation (2.2) for the three RE algorithms. T_{MD}, time replica takes to complete 500 time steps; T_{W}, synchronization time; T_{EX}, time to make a pairwise exchange; T_{find}, time to find/lock a partner; T_{file}, time to write/transfer files; T_{state}, time to update states after exchange; T, total time to completion.

	synchronous	asynchronous (decentralized) (s)	asynchronous (centralized) (s)
T_{MD}	71	71	71
T_{W}	2.8	0	0
T_{EX}	0.6	7.9	1.9
T_{find}	0	7.1	1.3
T_{file}	0.4	0.6	0.4
T_{state}	0.2	0.2	0.2
T	1003	631	811

in the following sections. The source code for implementing the different RE algorithms using the RE framework is available at: https://svn.cct.lsu.edu/repos/saga-projects/applications/async-re/.

(*a*) *Synchronous replica exchange*

In a homogeneous environment, the waiting time T_{W} for the synchronous RE implementation is primarily determined by the fact that the RE-Manager is only able to process one replica at a time, i.e. there is generally a delay between the start-up of one replica and the next. Since the post-processing of the replica ensemble takes longer than the delay between the start-up of the first and last replicas, the post-processing time determines the overall time spent waiting (T_{W}) for other replicas. Post-processing involves various state updates as well as the stage-out of the output file and requires on average 1.4 s per replica. Thus, T_{W}, which is defined for a pair of replicas, is 2.8 s. For an ensemble of 32 replicas (with 16 pairs) the delay between the first and the last replica transitioning to the done state adds up to 44.8 s.

T_{EX} comprises three sub-components: T_{find}, T_{file} and T_{state}. In this scenario T_{find} is 0 s owing to the fact that there is a fixed pairing. The updating and stage-out of the configuration files is observed to take approximately 0.2 s per replica; thus, T_{file} is 0.4 s per replica pair (the transfer of the input files is done sequentially). T_{state} is the time required by the RE-Manager to post a job description to the advert service. On average, T_{state} amounts to 0.1 s per replica, i.e. 0.2 s per replica pair. Thus, T_{EX} is $0 + 0.4 + 0.2 = 0.6$ s.

Although there are $N_{\mathrm{R}}/2$ concurrent pairs, the exchange at the RE-Manager is carried out sequentially; thus, the effective number of concurrently exchanging pairs (η) is 1. Substituting the above values in equation (2.2), we get

$$T = \frac{1}{p} \times \left[\left(71 \times \frac{128}{16} \right) + (0.6 + 2.8) \times \frac{128}{1} \right] = \frac{1}{p} \times 1003\,\mathrm{s}. \tag{4.1}$$

(*b*) *Asynchronous* (*decentralized*) *replica exchange*

A fundamental difference between the synchronous and asynchronous formulation of RE is in the synchronization barrier for the replicas before exchanges, i.e. the former has a barrier, the latter does not. As alluded to

earlier, we equate fixed replica pairing with the synchronous algorithm, and dynamic pairing with the asynchronous algorithm. Thus, although $T_W = 0$ for asynchronous formulations, this comes at the cost of a higher T_{find}, the time required to discover a new exchange partner. The lack of a synchronization barrier also leads to a more involved implementation to dynamically pair replicas.

T_{find} comprises two components: the pre-exchange management and the actual search part. As described, in the decentralized implementation of the asynchronous RE algorithm, the replicas are managed and exchanged by replica agents. The following pre-exchange steps are necessary. After a replica has completed its run, the agent updates the replica state at the advert service, which takes ≈0.1 s. The replica agent then retrieves the energy and temperature from the output file requiring $\approx 2 \times 0.2$ s. It then posts both values to the advert service, which takes $\approx 2 \times 0.1$ s. The total time for these pre-exchange operations is $0.1 + 0.2 \times 2 + 0.1 \times 2 = 0.7$ s.

The other component of T_{find} is the actual search time. When searching for a replica randomly, on average it takes $N_R/2$ attempts to find a replica that is in the `done` state. However, finding a replica in the `done` state is not enough to complete an exchange attempt. As there are several concurrent exchange attempts, contentions and conflicts can arise. Thus, a re-verify step must occur before the exchange is committed. During the re-verification, both replica agents that are involved in an exchange ensure that there has been a state change of either of the two replicas involved. Often, the re-verify step leads to an aborted exchange attempt. The exact number of necessary attempts is a random variable, determined by the number of replicas, the distribution of states and whether the attempt to find a replica is random or sequential. Empirical observations suggest that there are between two and four find and re-verify attempts before an exchange is successful. Also, replicas contend with each during this phase: the higher the numbers of replicas, the higher the contention. This means that T_{find} increases with a higher N_R. Specifically for the scenario with 32 replicas and random search, T_{find} is $2 \times N_R \times 0.1$ (where 0.1 is the typical time to set/get a value to/from the advert service), i.e. $6.4 + 0.7 = 7.1$ s.

T_{file} includes the process of posting, respectively, reading the states and temperatures to/from the advert service and writing a new configuration file: T_{file} is $0.1 \times 4 + 0.2 = 0.6$ s. T_{state} is the time required to update the state in the advert service, which is 0.2 s. In sum, T_{EX} is $7.1 + 0.6 + 0.2 = 7.9$ s. It should be noted that T_{EX} is highly dependent on the actual implementation. While the current implementation is kept simple on purpose, this value can be improved in a more sophisticated implementation.

As there are $N_R/2$ concurrent exchanges, η is 16. Substituting the above values in equation (2.2), we get

$$T = \frac{1}{p} \times \left(71 \times \frac{128}{16}\right) + (0 + 7.9) \times \frac{128}{16} = \frac{1}{p} \times 631\,\text{s}. \tag{4.2}$$

(*c*) *Asynchronous* (*centralized*) *replica exchange*

As in the decentralized implementation, the asynchronous centralized RE formulation does not require synchronization between *all* replicas in order for the transition of a replica pair from `running` to the `done` state and, thus,

$T_W = 0$ by definition. Using a centralized RE coordination mechanism, the time to find an exchange partner (T_{find}) can be reduced; however, this comes at a trade-off that there is some contention at the master. Also, we observed some delays at the BigJob-Agent during the start-up of the sub-jobs mainly because the BigJob-Agent is single-threaded and, thus, is busy processing other replicas after their termination. Specifically, the time to submit a replica pair to the BigJob-Manager, i.e. two replica sub-jobs, is on average 1.1 s for the asynchronous (centralized) formulation. This is 0.5 s longer than in cases without contentions at the BigJob-Agent—in these cases, the submission of two sub-jobs requires only 0.6 s.

In contrast to the synchronous case, T_{EX} for the asynchronous centralized case has a T_{find} component as replica pairs are dynamically determined and not fixed. T_{find} depends on the overall number of replicas (N_R), which determines the number of records the RE-Manager has to scan to find an available replica. If a random search is used, the RE-Manager must search through $N_R/2$ replicas on average, before it finds a partner. Although the search for a replica is random like in the decentralized implementation, there is no need for re-verifying, as owing to centralized control there are no conflicts in replica pairing, i.e. exchanges are made by the RE-Manager and only one exchange takes place at a time. An advert query for a replica state takes 0.01 s. Note that this is a factor of 10 less than the decentralized implementation. This is because, in the centralized case, all the values are located in the same advert directory, whereas, in the decentralized case, each replica's information is located in a distinct advert directory, so as to enable other replica agents to find it. For 32 replicas, the RE-Manager requests on average 16 other replica states before it finds a partner; thus, T_{find} is in total $0.01 \times 16 + 1.1 = 1.3$ s. Both T_{file} and T_{state} are the same as in the synchronous case, i.e. T_{EX} is thus $1.3 + 0.4 + 0.2 = 1.9$ s.

As in the synchronous case, the effective number of concurrently exchanging pairs is 1 because exchanges are sequentially carried out by the RE-Manager. Substituting the above values in equation (2.2), we get

$$T = \frac{1}{p} \times \left[\left(71 \times \frac{128}{16} \right) + (0 + 1.9) \times \frac{128}{1} \right] = \frac{1}{p} \times 811\,\text{s}. \tag{4.3}$$

5. Scale-up and scale-out: experiments and results

To evaluate the scaling properties of the different RE implementations, we conducted several experiments on TeraGrid and LONI resources. We initially increased the number of replicas on a single machine ('scale-up'); for the asynchronous centralized implementations, we varied the number of distributed machines used ('scale-out') for different replica counts. In this section, we discuss the experiments performed and analyse their results. For both scale-up and scale-out experiments, we find that, even though implementation details are important, the asynchronous RE algorithms have better scaling properties than the synchronous algorithms.

(a) Scale-up

(i) Experiments

To understand the scaling behaviour of the different RE formulations, we analyse T for replica counts from four, eight, 16, 32, 64, 128 and 256, making 16, 32, 64, 128, 256, 512 and 1024 exchanges, respectively. This fixes the ratio of the number of exchanges to the number of replicas (N_X/N_R) to 4. Each replica is configured to run 500 time steps before an exchange is attempted, and is allocated 16 processors. Experiments up to 64 replicas were performed on *QB*, while experiments with 128 and 256 replicas were done on *Ranger*; this is because 128 replicas would require 2048 cores and *QB* allows a maximum of only 2048 processors per job request. As getting a 2048-processor allocation involves extremely large waiting times in the queue, *Ranger* was used. We have normalized the data to factor in the difference in performance of *Ranger* and *QB*.

(ii) Results

As a consequence of the ratio of the number of attempted exchanges to the number of replicas being a constant, the number of times each replica is restarted to complete all exchanges remains constant; hence, comparison between different cases will reveal differences in the coordination cost (T_W and T_{EX}). The increase, however, is not uniform across the three implementations: it is largest for synchronous RE, and the least for asynchronous (decentralized) RE. We analysed T and the values of its components for 32 replicas in §4*a*; we use this analysis as the basis to understand the scale-up behaviour of the three formulations. The results of the experiments are depicted in figure 3. The increase in T as N_R increases is nonlinear for synchronous and asynchronous (centralized) RE and linear for asynchronous (decentralized) RE.

In the synchronous RE algorithm, there is an explicit synchronization of all replicas. As seen in table 1, T_W is a major component of the T. As N_R increases, the number of replicas that is required to synchronize at a given exchange step rises as well; consequently, the total coordination time at each exchange step increases. The cause for the increase in T_W is mainly algorithmic and arises owing to contention at the central RE-Manager process. Further, some implementation details, such as the fact that the current implementation of the RE-Manager is single-threaded, also affect T_W. Assuming $N_R = 64$ and, thus, $N_X = 256$, and using equation (2.2), T_W can be approximated to $(0.6 + 2.8) \times 256 = 870.4$ s. According to our model, T_W for $N_R = 32$ is 485 s, which is consistent with the change in T observed in the experiments depicted in figure 3. A similar analysis was performed for different replica and exchange counts, and the values obtained are in agreement with the empirical data in figure 3.

For asynchronous (decentralized) RE, T_W is 0. T_{find} has a strong N_R dependence. The finding operation involves several queries to the advert service; T_{find} can be approximated as follows: $2 \times N_R \times 0.1$ (where 0.1 is the typical time to set/get a value to/from the advert service). From §4*b* and equation (4.2), it can

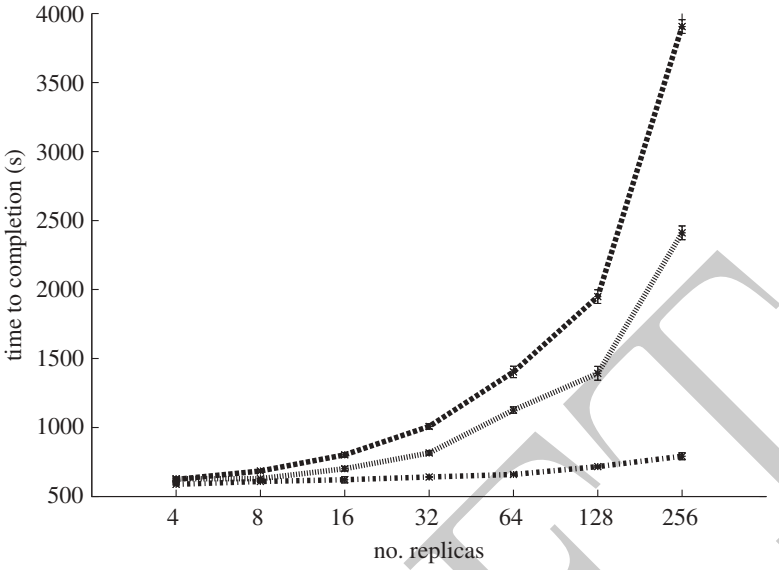

Figure 3. Scale-up performance for 4–256 replicas. The graph shows the runtimes for the different RE implementations. The ratio between the number of exchanges and the number of replicas is kept constant. Each replica is assigned 16 processors and run 500 time steps. The asynchronous decentralized RE implementation shows the best scaling behaviour. Both centralized RE versions scale less well mainly because of the limitations of the single master, which becomes a bottleneck. Dashed line, synchronous; dotted line, asynchronous (centralized); dashed-dotted line, asynchronous (decentralized).

be derived that, for 64 replicas, T_{find} is approximately $0.7 + 2 \times 64 \times 0.1 = 13.5\,\text{s}$. The difference in the coordination cost ($T_{\text{EX}} \times N_{\text{X}}/(N_{\text{R}}/2)$) between an N_{R} value of 64 and 32 is 45 s, which accounts for the bulk of the increase in overall T. The observed performance behaviour can be attributed both to the algorithm, i.e. the used random search method, as well as to specific implementation details, such as the usage of the central advert service.

In contrast to asynchronous (decentralized) RE, T_{find} in asynchronous (centralized) RE is only weakly dependent on N_{R}; however, since the RE-Manager makes the exchanges serially, we still see an increase in the time to completion with increasing N_{R}. For $N_{\text{R}} = 64$ and $N_{\text{X}} = 256$, using equation (4.3) leads to a new coordination ($T_{\text{W}} + T_{\text{EX}}$) time of $(0 + 1.9) \times 256 = 486.4\,\text{s}$, up from 243.2 for $N_{\text{R}} = 32$. This change accounts for a large component of the difference in T as N_{R} goes from 32 replicas to 64 replicas, as shown in figure 3. The actual difference observed in T from 32 to 64 replicas is 311 s. We have verified this scale-up model for different numbers of replicas and found it to be consistent with the data in figure 3.

As N_{R} increases and as more nodes of a machine are used, we find that some of them tend to perform differently, thus influencing the time to completion of different replicas; however, these fluctuations are small and do not mask the dominant increase with increasing values of N_{R}—which is primarily due to increased synchronization times.

(*b*) *Scale-out: asynchronous replica exchange* (*centralized*)

(i) *Experiments*

In this section, we scale-out to two and four distributed resources to evaluate the performance of the asynchronous (centralized) RE. We used the TeraGrid and/or LONI resources, namely *QB, Eric, Louie* and *Oliver*. The experiments were conducted with eight, 16 and 32 replicas that make 32, 64 and 128 exchanges, respectively. These exchanges were repeated on one, two and four machines while distributing an equal number of replicas on each machine. It is important to note that all experiments were conducted using four big-jobs, irrespective of the number of machines used, because varying the number of big-jobs, or the ratio of replicas per big-job, affects the overall performance. Another important point to note is that only the experimental runs in which all the four big-jobs become active within 30 s of each other on submission to the resource manager are considered and included in the results. This is to remove the effect of queue wait time and to get an accurate assessment of the runtime. Each experiment was repeated five times.

(ii) *Results*

We analyse the performance of the asynchronous (centralized) RE using a varying number of distributed resources (one, two and four) while keeping the number of replicas constant. After factoring in the performance advantage owing to the additional big-jobs/BigJob-Agents, the analysis provided in §4*c* can be used to understand the behaviour using the model developed in §2.

Even though, in a distributed scenario, T_{MD} can be different on different machines, the resources used in this experiment are mostly homogeneous in performance; thus, T_{MD} is not affected. Also, depending on the physical location and other network-related aspects, T_{EX} can be affected. Because T_{W} is 0 s in asynchronous RE, the main component that influences T is T_{EX}, which comprises T_{find}, T_{file} and T_{state}. Finding exchange partners and book-keeping involves several communications via the advert service and some local processing. Although the resources are homogeneous and all belong to LONI, they are geographically distributed. This has a minor impact on the latency observed in the communication with the advert service: from *QB*, *Eric* and *Oliver*, the creation of an advert entry takes 0.01 s, from *Louie* it requires 0.02 s. As the variation in these times is very small, the impact on T can be neglected. We observed that T_{file}—i.e. the time to transfer the configuration file to a replica—has a bigger impact on T.

The results obtained from the scale-out experiments are plotted in figure 4; note that the scale of the y-axis is much smaller than in figure 3. The variations in T are small, ranging from 10 to 20 s. The fluctuations are all within the 30 s tolerance imposed by the experimental method.

For eight replicas conducting 32 exchanges on one machine, T is 570 s; if these replicas are distributed equally over two machines, T increases slightly to 586 s. In total, the observed overhead of distribution is quite small and essentially acceptable (with at worst 5% of the overall runtime). As explained earlier, the main source of these overheads is the necessary remote file transfers. Each remote file transfer takes 0.38 s, while a local transfer requires only 0.01 s, i.e. 0.37 s less.

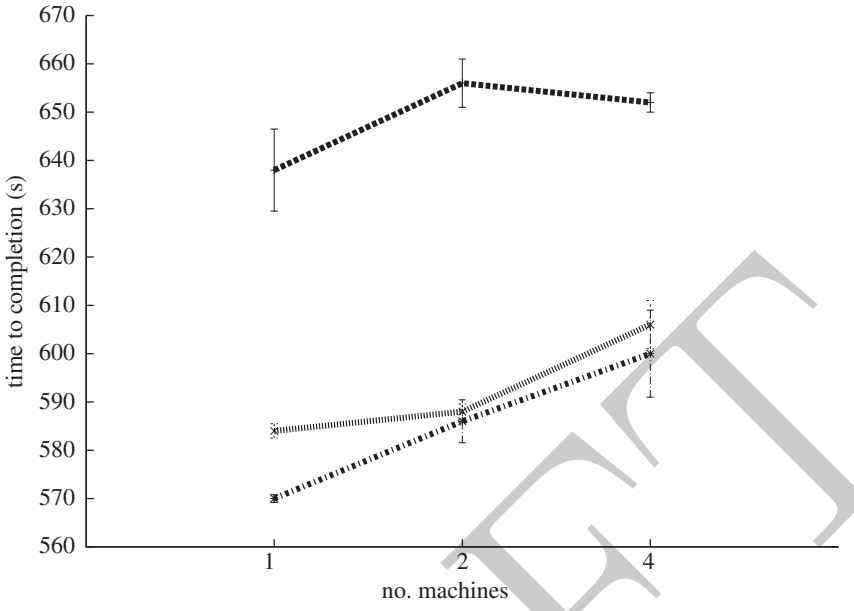

Figure 4. Scale-out performance for eight, 16 and 32 replicas, asynchronous (centralized). The experiments were done on LONI resources and repeated at least five times. The error bars denote standard error. As the number of machines increases, the time to completion increases in general mainly because of higher exchange costs (T_{EX}) caused by, for example, remote file copies. Dashed line, 32 replicas; dotted line, 16 replicas; dashed-dotted line, eight replicas.

For the eight replicas, 32 exchanges and two machines configuration, half of the replicas are located on a different resource from the RE-Manager, i.e. the input file must be transferred to the remote resource in these cases. On average, when two machines are used, one of the necessary input files for an exchange requires a remote copy; therefore, for the eight replicas, 32 exchange configurations, 32 remote transfers are necessary. Thus, the additional overhead in T_{file} can be approximated to $0.37 \times 32 = 12$ s. In figure 4, this accounts for the difference, to within error bars, between the T values of one and two machines (for the eight-replica configuration).

As mentioned, we keep the ratio of the number of replicas to the number of exchanges a constant. Thus, as the number of replicas increases the time spent in the exchanges as a fraction of the overall time becomes less; thus lessening further any effects of the small increase due to distributed exchange. This explains why at low replica counts there is a small (5%) increase in T as the number of distributed resources increases from one to four, but effectively there is no change in T when going from one to four machines when N_{R} is 32.

Note that the fluctuations are in the range of 10–20 s. As mentioned earlier, the big-jobs became active within 30 s of each other, thus the fluctuations are within the tolerance of the experimental design. Additionally, for the 32-replica configuration, we see a larger fluctuation in T on one machine than on two and four machines. There are several reasons that can be attributed to the

decrease in the fluctuation in going from one to two (and four) machines; however, the dominant reason for the high fluctuation for one machine is likely to be finite sampling—recall that experiments were repeated only five times. We plan to further the number of experimental samples, and at larger scales to analyse their behaviour accurately. Nonetheless, from these results, we can say that our RE framework is capable of scaling out without acutely affecting the performance.

Our model and data from figure 4 might only suggest additional file transfer costs, but in a typical distributed scenario, with machines on different networks, physical locations, cost of distributed exchanges and heterogeneity may also play a role in affecting the performance. We intend to examine these factors in the future; obviously a comparison with other RE implementations will also be performed.

6. Conclusion

Following on from earlier theoretical explorations [7,8], in this paper, we investigate *traditional* and *advanced* RE algorithms at unprecedented scales. An important motivation for this work has been to implement a framework that supports multiple RE algorithms; it is the aim that the RE framework uses general purpose and standard capabilities available on production infrastructures, such as the TeraGrid and LONI. Additionally, our framework uses a flexible pilot-job implementation, which enables effective resource allocation for multiple replicas.

Results shown in figures 3 and 4 indicate that using algorithmic formulations that impose less tight coordination constraints enable both good scale-up and scale-out behaviour. Algorithms based on asynchronous coordination are typically more difficult to implement than synchronous ones; however, we find that, even with a simple, non-optimized prototype of the RE framework, the advantages of asynchronous formulations soon outweigh the synchronous formulations, i.e. as N_R increases, the performance of asynchronous RE beats that of the synchronous RE.

Our analysis shows that a fundamental trade-off is between the lower cost of replica synchronization at the exchange stage that asynchronous formulations provide, versus the higher cost of permitting dynamic replica pairing. In an attempt to investigate an optimal trade-off between these factors, and to demonstrate the advantages of asynchronous RE, we implemented a centralized version of the asynchronous RE with a lower cost of dynamical pairing than in the decentralized implementation. Our initial results show promising scale-out behaviour, but more work is required to separate and understand the fundamental algorithmic advantages from implementation-specific issues.

This work is part of the Cybertools (http://cybertools.loni.org) project and is primarily funded by NSF/LEQSF (2007-10)-CyberRII-01. Important funding for SAGA has been provided by UK EPSRC grant no. GR/D0766171/1 (via OMII-UK) and HPCOPS NSF-OCI 0710874. This work has also been made possible thanks to computer resources provided by TeraGrid TRAC TG-MCB090174 and LONI resources. We thank Nayong Kim (LSU) and Tom Bishop (Tulane) for their useful comments and suggestions. Computing resources were made possible via TRAC award TG-MCB090174.

References

1 Hansmann, U. H. E. 1997 Parallel tempering algorithm for conformational studies of biological molecules. *Chem. Phys. Lett.* **281**, 140–150. (doi:10.1016/S0009-2614(97)01198-6)

2 Sugita, Y. & Okamoto, Y. 1999 Replica-exchange molecular dynamics method for protein folding. *Chem. Phys. Lett.* **314**, 141–151. (doi:10.1016/S0009-2614(99)01123-9)

3 Woods, C. J. *et al.* 2005 Grid computing and biomolecular simulation. *Phil. Trans. R. Soc. A* **363**, 2017–2035. (doi:10.1098/rsta.2005.1626)

4 Luckow, A., Lacinski, L. & Jha, S. 2010 Saga bigjob: an extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Proc. 10th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, Melbourne, Australia, 17–20 May 2010*. Washington, DC: IEEE Computer Society.

5 Luckow, A., Jha, S., Kim, J., Merzky, A. & Schnor, B. 2009 Adaptive distributed replica-exchange simulations. *Phil. Trans. R. Soc. A* **367**, 2595–2606. (doi:10.1098/rsta.2009.0051)

6 Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. & Teller, E. 1953 Equation of state calculations by fast computing machines. *J. Chem. Phys.* **21**, 1087–1092. (doi:10.1063/1.1699114)

7 Li, Z. & Parashar, M. 2007 Grid-based asynchronous replica exchange. In *GRID '07: Proc. of the 8th IEEE/ACM Int. Conf. on Grid Computing, Austin, TX, 19–21 September 2007*, pp. 201–208. Washington, DC: IEEE Computer Society.

8 Gallicchio, E., Levy, R. M. & Parashar, M. 2008 Asynchronous replica exchange for molecular simulations. *J. Comp. Chem.* **29**, 788–794. (doi:10.1002/jcc.20839)

9 Phillips, J. C. *et al.* 2005 Scalable molecular dynamics with NAMD. *J. Comp. Chem.* **26**, 1781–1802. (doi:10.1002/jcc.20289)