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

Novel submission modes for tightly coupled jobs across distributed resources for reduced time-to-solution

Promita Chakraborty, Shantenu Jha and Daniel S. Katz

Phil. Trans. R. Soc. A 2009 **367**, 2545-2556
doi: 10.1098/rsta.2009.0054

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

Phil. Trans. R. Soc. A (2009) **367**, 2545–2556
doi:10.1098/rsta.2009.0054

Novel submission modes for tightly coupled jobs across distributed resources for reduced time-to-solution

By Promita Chakraborty[1,2,*], Shantenu Jha[1,2,3,*] and Daniel S. Katz[1]

[1]*Center for Computation & Technology, and* [2]*Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA*
[3]*e-Science Institute, Edinburgh EH8 9AA, UK*

The problems of scheduling a single parallel job across a large-scale distributed system are well known and surprisingly difficult to solve. In addition, because of the issues involved in distributed submission, such as co-reserving resources, and managing accounts and certificates simultaneously on multiple machines, etc., the vast number of high-performance computing (HPC) application users have been happy to remain restricted to submitting jobs to single machines. Meanwhile, the need to simulate larger and more complex physical systems continues to grow, with a concomitant increase in the number of cores required to solve the resulting scientific problems. One might reduce the demand on load per machine, and eventually the wait-time in queue, by decomposing the problem to use two resources in such circumstances, even though there might be a reduction in the peak performance. This motivates a question. Can otherwise monolithic jobs running on single resources be distributed over more than one machine such that there is an overall reduction in the time-to-solution? In this paper, we briefly discuss the development and performance of a parallel molecular dynamics code and its generalization to work on multiple distributed machines (using MPICH-G2). We benchmark and validate the performance of our simulations over multiple input datasets of varying sizes. The primary aim of this work, however, is to show that the time-to-solution can be reduced by sacrificing some peak performance and distributing over multiple machines.

Keywords: job submission paradigm; tightly coupled distributed performance; scheduling

1. Introduction

The need to simulate larger and more complex systems continues to grow, with a concomitant increase in the number of cores required to solve the resulting scientific problems. The number of processors typically available for specific compute jobs on a given machine has been increasing and will continue to

* Authors and addresses for correspondence: Center for Computation & Technology, Louisiana State University, Baton Rouge, LA 70803, USA (promita@cct.lsu.edu; sjha@cct.lsu.edu).

One contribution of 16 to a Theme Issue 'Crossing boundaries: computational science, e-Science and global e-Infrastructure I. Selected papers from the UK e-Science All Hands Meeting 2008'.

increase. But often, problem sizes of interest are so large that they cannot be run on any available single resource. Additionally, using an increasing number of processors for a tightly coupled simulation has its own challenges, as message passing, the dominant paradigm for developing tightly coupled applications, is reaching its limits of scalability—at least for some applications.

Significant effort continues to be invested in *scaling-up* the performance of applications. Critical as this endeavour is, there are limitations, because beyond some point additional efforts will have diminishing returns. Not surprisingly, the metric of maximum concern to the bulk of scientists is not *peak performance* (say measured in Tflops), but total time-to-solution (T_s). To a first approximation, the total T_s can be decomposed into run-time (T_{run}) and wait-time for requested resources to become available (T_{wait}) (this is often just the queue wait-time). Ironically, although much attention has been focused on reducing T_{run}, little attention, at least compared with the former, has been devoted to reducing the latter.

Jobs with larger processor count requirements are typically associated with longer wait-times than jobs with smaller processor counts. Additionally, the wait-time of a job in a queue depends on the estimated job duration (maximum wall-clock time requested), as well as the number of processors needed. At present, most users tend to submit jobs, however large, to a single machine, as opposed to a set of machines over a distributed environment. This has two main advantages: (i) the job runs faster and (ii) it consumes less CPU hours than that in a distributed environment. The disadvantages are: (i) the job requires all resources to be available on a single machine, instead of distributing the load, and (ii) the job experiences larger wait-time in queue than when it is distributed across several machines.

Interestingly, although the dominant computational model for high-performance computing has involved using only a single resource, with advances in grid technologies, high-performance simulations over multiple resources are now feasible (Manos *et al.* 2008). In this case, decomposing a simulation to use two or more resources is a natural response to the failure of a given system to meet peak demand; thus such decomposition can be referred to as an example of needful decomposition. However, users normally consider multiple machines only when the number of nodes required by the job is greater than what a single machine can provide; needful decomposition is not the only situation under which multiple, distributed resources can or should be used.

This motivates some questions. Can otherwise monolithic jobs running on single resources be distributed over more than one machine such that there is an overall reduction in the time-to-solution? What are the challenges in doing so? What is the performance? And what system-level support is required to make such decomposition strategies meaningful/usable?

Although the capability to launch a single tightly coupled task over two different resources exists, different workloads and queueing systems make using a distributed job challenging in practice. As we will show in §3, tightly coupled simulations, such as those using MPICH-G2, go into essentially a stalled state, waiting for all resources to become available. A commonly used approach is that of invoking the service of a co-scheduler, such as the Highly-Available Resource Co-allocator (HARC; http://www.cct.lsu.edu/harc/) or the Grid Universal Remote co-allocator (GUR; http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/TGIA64LinuxCluster/Doc/coschedule.html). Although the specific algorithms and implementation details of the co-schedulers differ, most

co-schedulers require the prior permission of system administrators to make a reservation. We propose a solution that, with some minor caveats, is essentially in user space and does not require the pre-negotiation of advance reservation rights.

In this paper, we will demonstrate the feasibility of decomposition of a tightly coupled simulation, to support novel submission modes as a strategy to reduce the effective time-to-solution; we refer to such decomposition as a type of opportunistic decomposition. We will briefly discuss the development and performance of a parallel molecular dynamics (MD) code and its generalization to work on multiple distributed machines (using MPICH-G2). We benchmark and validate the performance of our application over multiple input datasets of varying size. The primary aim of this work, however, is to show that jobs can finish sooner (i.e. lower time-to-solution) by an appropriate configuration of resource requests—defined as a selection of the number of processors and machines, even though the peak performance might be poorer. Or, formulated in a way that would make Zeno[1] proud: how to finish sooner by running slower!

The outline of this paper is as follows. In §2, we describe our parallel MD code, the models and test-beds used. In §3 we narrate and depict the control flow for job submission to a single machine as well as multiple machine(s), and highlight the difference in environments. In §4 we present results—on job runs, actual wait-times taken and wait-time predictions from Batch Queue Predictor (BQP)—formulate the necessary equations, and show that *often* the time-to-solution is lower in the distributed mode, in spite of higher run-time values. Our conclusions and some discussion related to other work are presented in §5.

2. Tightly coupled simulations: background information

Owing to the frequent communication requirement, MD codes are typically deployed on tightly coupled machines. It is often difficult to scale message passing codes to beyond $O(1000)$ processes. Inter-process communication bandwidth requirements are not very large, but MD codes are latency sensitive. Thus, although MD simulations are not naively suitable for distributed parallelism, our aim is to see whether new ways of 'distributing' such applications can yield better performance, not only in terms of speed, but also in terms of other issues such as queue wait-time, load balance, etc., and to see whether current grid implementations support this paradigm.

We developed a parallel MD code (Chakraborty & Jha 2008) using MPICH-G2 and C++, based on the serial MD code `Mindy`[2]. Our domain decomposition was not complex; we simply divided the dataset and the number of atoms among the available processors. We studied the performance of this parallelized and distributed MD code initially on LONI machines (Bluedawg, Zeke and Ducky clusters) and then on TeraGrid machines (NCSA and SDSC TeraGrid clusters). LONI (Louisiana Optical Network Initiative) is a Louisiana-wide network of supercomputers connected by light paths, and is connected to the TeraGrid (TG). The NCSA (National Center for Supercomputing Applications) and SDSC (San Diego Supercomputer Center) TG clusters are IA-64 machines.

[1]An ancient Greek philosopher who formulated paradoxes that defended the belief that motion and change are illusory.

[2]Available from the `NAMD` website http://www.ks.uiuc.edu/Research/namd.

We used the BQP (http://spinner.cs.ucsb.edu/batchq/invbqueue.php) to get an estimate of the wait-time of jobs on NCSA and SDSC TG clusters. Currently, BQP tools provide two types of estimates for a given set of job characteristics: (i) they can estimate a statistical upper bound on job wait-time in the queue prior to execution and (ii) given a start deadline, they can calculate the probability that the job begins execution by the deadline. We used the second option. It is to be noted that BQP predictions are for single machines only.

In order to avoid confusion and to retain clarity, we define some terminology that we use in this paper. By *job submission to a single machine*, we mean that the user specifies that the whole job is to be submitted to a specific single machine (stand-alone) available on the grid. By *job submission to multiple machines*, we mean that the user specifies that a single job is to be divided (into sub-jobs) among a particular set of machines available. Thus a *sub-job* is a part of a single job that is submitted to one of a set of machines, not a smaller piece of job submitted to a node of a machine. *Time-to-solution* (T_s) is a measure of the sum of run-time and queue wait-time for a job.

3. Control flow for job submission: single and multiple machines

Often, a job is remotely submitted through Globus, using a Resource Specification Language (RSL) submission file that contains information about the specific machine(s) for job submission, the number of processors needed, the maximum wall-clock time, etc. Once the job is submitted, control passes to the Globus Resource Allocation Manager (GRAM). The RSL file is parsed to get the resource(s) information. The GRAM gatekeeper interacts with the job schedulers and submits the jobs to the respective scheduler(s). Some examples of job schedulers are LoadLeveler (on the LONI IBM P5 clusters) and PBS (on Queen Bee). If a job is submitted to a single machine, the local scheduler adds the job to the queue, where it waits until requested nodes become available. The local resource manager (RM) assigns the available processors to the waiting jobs, often using first in, first out or other fairness principles. When a message passing interface (MPI) job starts executing, an MPI environment/communicator is created and initialized with N processes. If the job does not finish execution within the specified wall-clock time value provided by the user in the RSL file, the RM kills it. So, users should have a good idea of the time the job might take to execute. On the other hand, if the wall-clock time value provided is too large, the jobs may stay in the queue longer.

If a job is submitted to multiple machines using a single submission (with a single RSL file), the process is more complicated. The code needs to be compiled using MPICH-G2. The GRAM gatekeeper interacts with all the machines requested by the user, and the respective job schedulers put the sub-jobs in the respective queues. Here, the RMs might not be able to allocate processors to all the sub-jobs simultaneously. When the first set of resources is allocated, the sub-job in that machine initializes the MPI environment, and waits for responses from other machines. While the sub-job is waiting for other sub-jobs to start, it is officially in a run state from the point of view of the local RM, and if it waits for too long it will be killed. Hence, it is important that the sub-jobs in all the different machines are started simultaneously. Figure 1*a*,*b* shows the control flow for job submission to a grid for a single machine and for multiple machines, respectively.

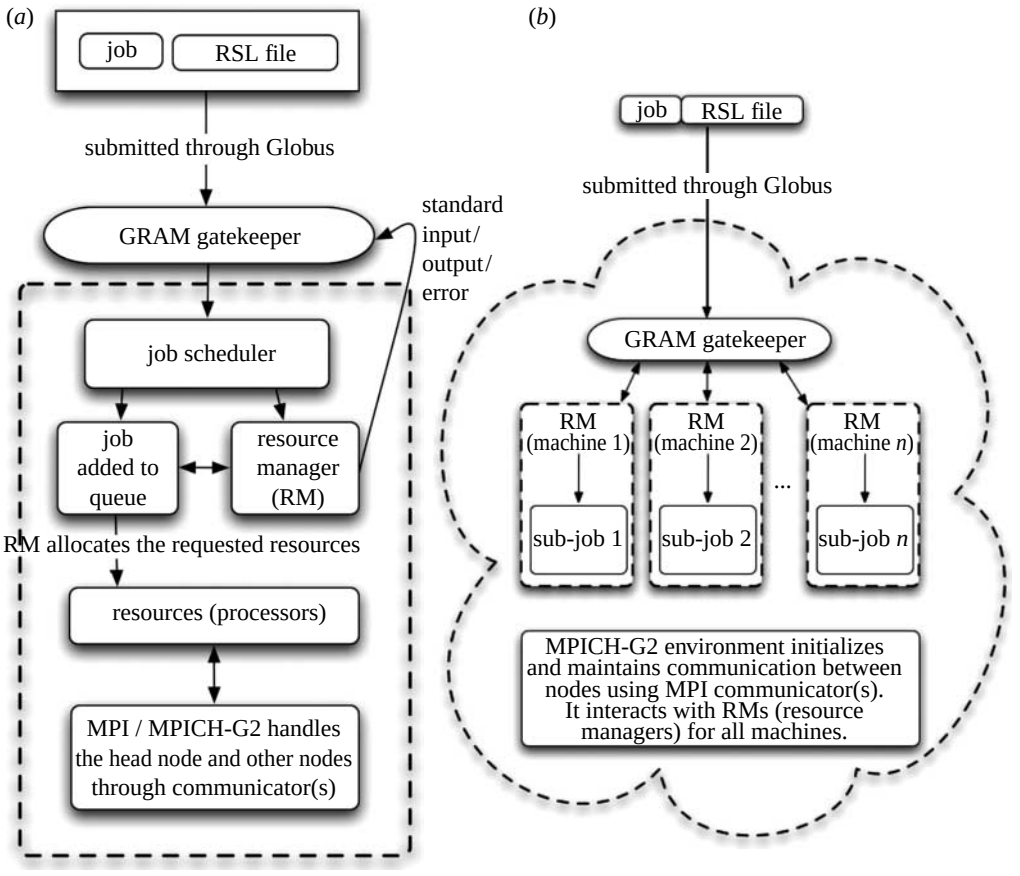

Figure 1. (*a*) Control flow for job submission to a single machine; the dotted rectangle represents a single machine. MPI/MPICH-G2 creates an MPI Communicator World to keep track of the communication between different processors in the machine. (*b*) Control flow for job submission to multiple machines; each dotted rectangle represents a single machine. By communicating with the RM and GRAM, the MPICH-G2 environment ensures that the sub-jobs on all the distributed machines are allocated the required number of processors.

4. Results and discussion

We used two physical systems as models for our experiments: alanine and bacteriorhodopsin (BrH). Alanine is a 66-atom polypeptide and BrH is a crystal structure containing 3762 atoms.

(*a*) *Performance results and resource usage*

(i) *Distributed MD performance results on LONI*

Initially, our aim was to test only the feasibility of running a single parallel job across multiple machines, and to quantify the performance of the job. We executed the parallel `Mindy` code on three IBM AIX P5 clusters, Bluedawg (BL), Ducky (DU) and Zeke (ZE), on LONI. Here, we provide the data and a

Table 1. Performance on single machines using eight processors in each case.

(*a*) CPU seconds for BrH model

time steps	T_r(BL)	T_r(DU)
100	0.2	0.2
1000	2.4	2.4
10 000	23.9	23.9
100 000	232.9	235.3
1 000 000	2518.6	—

(*b*) CPU seconds for alanine model

time steps	T_r(BL)	T_r(DU)	T_r(ZE)
50 000	5.6	5.6	6.0
100 000	11.1	11.2	12.2
1 000 000	110.6	111.9	121.1
10 000 000	1098.7	1113.97	1207.0
100 000 000	—	10860.1	11054.9

Table 2. Performance comparison as measured by the total computational time (CPU seconds required) on Ducky when using different numbers of processors for 10^5 and 10^6 time steps (for BrH model). (*Px*, number of processors; T_r, run-time.)

		100 000 time steps		1 000 000 time steps	
Px	no. of nodes	T_r (s)	CPU seconds	T_r (s)	CPU seconds
12	2	118.8	1426.1	1249.6	14 995.2
24	3	46.8	1123.7	487.96	11 711.0
32	4	36.7	1175.5	374.3	11 977.0
40	5	32.5	1299.6	333.3	13 331.2
48	6	29.6	1420.0	299.2	14 358.2
56	7	28.7	1606.1	288.99	16 183.3

quantitative analysis for it, and we estimate the performance degradation associated with distributed job submission. On LONI the wait-times for our jobs were trivial—of the order of a few minutes (although there were cases when jobs were killed while waiting in queue for a long time, due to the system being overloaded). Hence, we concentrated on run-time, T_r, i.e. CPU seconds used, only. The total amount of CPU time consumed for a simulation was calculated using the formula: CPU seconds = time taken × number of processors, where time taken is measured in seconds. Table 1*a*,*b* highlights the application's performance on individual LONI machines; Table 1*a* shows the performance on BL and DU separately when the code runs on eight processors (on a single node) using the BrH model. Table 1*b* shows the same for the alanine model, with machines BL, DU and ZE. The table shows that the processing time is proportional to the number of time steps over which the simulation is run. Thus, at the largest time

Table 3. A comparison of the time taken (in seconds) by DU alone, with the time taken on DU and ZE (combined) for different processor configurations (for BrH model). (PD, performance degradation; *Px*, number of processors; T_r, run-time.)

	DU			DU+ZE			
time steps	*Px* (*x*)	T_r (s)	CPU seconds	*Px* (*x*+*x*)	T_r (s)	CPU seconds	PD (%)
1000	8	2.4	19.5	4+4	3.0	23.8	22.05
10 000		23.9	191.5		29.3	234.3	22.35
100 000		235.3	1882.7		288.5	2308.1	22.59
1000	16	0.7	11.1	8+8	0.8	13.4	20.72
10 000		7.7	123.7		9.4	149.6	20.94
100 000		76.2	1219.0		92.6	1481.9	21.57
1000	32	0.1	3.7	16+16	0.1	3.8	2.70
10 000		3.6	116.0		3.9	124.0	6.90
100 000		36.7	1175.5		40.1	1282.7	9.12

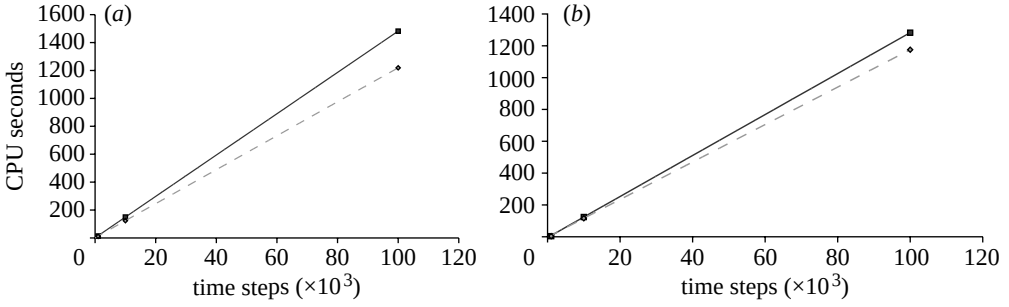

Figure 2. A performance comparison of the time taken (in seconds) by a single machine versus two machines at a time (for BrH) on LONI: (*a*) 16 (DU; diamonds) versus 8+8 (DU+ZE; squares) processors; and (*b*) 32 (DU; diamonds) versus 16+16 (DU+ZE; squares) processors.

step values, it is valid to assume that initial/transient/set-up effects are no longer relevant. The table also shows that the performance of BL and DU are essentially the same, to within a few per cent.

Having established that start-up effects no longer contribute at 10^5 time steps, we measured the computational time required (T_r) as the processor count was varied on DU (table 2). As shown in the table, for the BrH model, there is a speed-up as the processor count is increased up to 24 (although not a slope of unity); after which, increasing the processor count leads to a net increase in computational cost.

Table 3 compares the computational time (CPU seconds) for the BrH model when run on DU (stand-alone) with the computational time taken when run on DU and ZE (combined) for a range of different processor count configurations. We see that the performance in the distributed environment degrades between 2 and 22 per cent (approx.) compared with the performance on a single machine. Furthermore, on increasing the number of processors, the performance improves and the degradation percentage decreases (figure 2*a*,*b*). This raises interesting

Table 4. A comparison of the time taken (in seconds) by NCSA and SDSC TeraGrid cluster machines for different processor configurations (for BrH model). (Px, number of processors; T_r, run-time; T_w, wait-time.)

	NCSA			SDSC			NCSA + SDSC		
time steps	Px	T_r	T_w	Px	T_r	T_w	Px	T_r	T_w
1000	8	3.67	306	8	3.83	450	4+4	6.74	78
10 000		35.77	306		38.12	456		65.83	78
100 000		352.85	306		371.23	456		657.20	78
1000	16	3.01	1050	16	3.67	1272	8+8	8.74	510
10 000		27.33	1062		35.77	1266		85.83	516
100 000		240.96	1062		352.85	1278		857.20	516

possible usage modes, i.e. instead of spending extra hours in queue waiting for resource allocation, whether parallel code should be distributed over multiple machines.

(ii) *Distributed MD performance results on TeraGrid with emphasis on wait-time*

After testing on LONI, we performed similar experiments on the TeraGrid. But unlike on LONI (where wait-times were often negligible), here we also recorded the wait-time taken by the jobs. The TeraGrid environment proved to be more challenging than LONI owing to the size of the virtual organization and multiple resource providers with separate policies; LONI resources used for this experiment, by contrast, were essentially identical—in terms of both policy and infrastructure.

Table 4 compares the computational time (in seconds) for the BrH model, when run on NCSA and SDSC clusters (stand-alone), with the computational time taken when run on both the NCSA and SCSC clusters (combined), for a range of different processor count configurations. We observe that the run-time of jobs is proportional to the number of time steps calculated, as on LONI. Roughly, for a 10-fold increase in time steps, the run-time for the jobs increased 10-fold. However, the most interesting thing to note is the behaviour of the wait-time. Although the wait-times remained more or less constant for increasing time steps (i.e. for increased run-times), they increased dramatically for increasing number of processors. Moreover, jobs run on NCSA and SDSC combined had significantly shorter wait-time than those run on either, alone. Hence, we concluded that an overall reduction in the time-to-solution in distributed mode is feasible, in spite of the fact that stand-alone machine run-times were far less than that in the combined (distributed) scenario.

(iii) *Opportunistic distributed resource usage* (*BQP*)

For single resources, the BQP (http://spinner.cs.ucsb.edu/batchq/invbqueue.php) provides users with the probability of running a job on a number of nodes within a certain pre-selected deadline. As BQP predictions are for single

Table 5. BQP data for NCSA and SDSC TeraGrid clusters (for Queue=dque for both). The BQP user provides the value of the number of processors to be used and a best estimate for the run-time required; based upon that, BQP returns a value of the wait-time on a resource for different probabilities. We choose a value of approximately 0.6; our results are not sensitive to the specific value of probability chosen. (*Px*, number of processors; T_r, run-time; T_w, wait-time (times measured in seconds).)

Px	estimated T_r	probability	approx. T_w (NCSA)	approx. T_w (SDSC)
8	900	0.61	530	780
	1800	0.61	2130	4200
	3600	0.61	2130	4200
	7200	0.61	5150	8100
16	900	0.61	2130	4200
	1800	0.61	2130	4200
	3600	0.61	5150	8100
	7200	0.61	5150	8100

machines only, we first estimated the wait-time in queue separately for each of the machines involved using the BQP tool. We then combined the results to get the approximate wait-time for the two machines—the NCSA and SDSC TG clusters (table 5). We then used this as estimated average wait-time, in order to attempt to reduce the total time-to-solution over multiple resources. The use of BQP keeps the entire scheduling decision and resource co-allocation in user space (as opposed to when using advanced reservations).

Table 5 shows the estimated wait-time as calculated by BQP for the NCSA and SDSC TG clusters. We see that, in some cases, higher run-times do not cause the wait-times to increase. This was also supported by actual run results in table 4, where the wait-time does not increase with the number of time steps. Now, if we consider jobs with run-time of 900 s from table 5, we see that for eight processors the approximate wait-time is 530 s; we got the same value for four processors (not shown here); but for 16 processors, the wait-time jumps to 2130 s. Hence, if we distribute this job into two eight-processor sub-jobs across the two machines, the wait-time is reduced by a factor of 4, although the run-time might increase by 10–20 per cent as was observed earlier (tables 3 and 4). So, the net time-to-solution decreases in the distributed mode in such cases.

Overall, in designing a multiple-machine job submission paradigm, BQP prediction can help us in two ways. First, it can help us to decide whether it is beneficial to distribute (as in the case discussed above) and second, if beneficial, it tells us what the expected wait-time in the distributed paradigm is.

(*b*) *Quantitative decision making*

Let $T_{\mathrm{wait}}(i,q)$ be the wait-time and $T_{\mathrm{run}}(i,q)$ be the run-time for a job on machine i using q processors. If $T_s(i,q)$ is the estimated time-to-solution of a job on machine i using q processors, then $T_s(i,q) = T_{\mathrm{wait}}(i,q) + T_{\mathrm{run}}(i,q)$. For jobs spread over multiple machines: $T_s(i,q_i;j,q_j;\ldots) = \mathrm{Max}[T_{\mathrm{wait}}(k,q_k) + T_{\mathrm{run}}(k,q_k)]$, k in $\{i,j,\ldots\}$, where the Max is computed over all the machines used.

We have obtained the wait-time in two different ways: directly from job runs, and from BQP data. BQP provides an estimate of $T_{\mathrm{wait}}(i,q)$, which varies quite significantly not only with i (i.e. different resources have different load levels at a given time), but also for different values of q for the same i (for example, $T_{\mathrm{wait}}(\mathrm{mymachine},16) \ll T_{\mathrm{wait}}(\mathrm{mymachine},32)$). Thus by estimating the values of $T_{\mathrm{run}}(i,q)$, the optimal values of i and q can be determined so as to find the minimum T_{s}.

For the single machine i (either NCSA or SDSC) using 16 processors, let $T_{\mathrm{s}}(i,16) = T_{\mathrm{run}}(i,16) + T_{\mathrm{wait}}(i,16)$; and for the two machines (NCSA and SDSC), let $T_{\mathrm{s}}(\mathrm{NCSA},8;\mathrm{SDSC},8) = T_{\mathrm{run}}(\mathrm{NCSA},8;\mathrm{SDSC};8) + T_{\mathrm{wait}}(\mathrm{NCSA},8;\mathrm{SDSC},8)$. From data collected on TeraGrid and LONI, we obtained

$$T_{\mathrm{run}}(i,8;j,8) \approx aT_{\mathrm{run}}(i,16), \quad \text{where } 1.1 \le a \le 1.2,$$

for 10–20 per cent performance degradation, where a is a measure of performance degradation in switching from one machine to multiple machines.

Based upon empirical (reproducible) observation on the TG, we find

$$T_{\mathrm{wait}}(i,16) >> T_{\mathrm{wait}}(\mathrm{NCSA},8;\mathrm{SDSC},8),$$

which often holds true for a range of processor counts, and i could be either the NCSA or SDSC IA-64. For example, for an estimated run-time of 900 s for 16 versus 8+8 processors,

$$T_{\mathrm{wait}}(\mathrm{NCSA},8;\mathrm{SDSC},8) \approx bT_{\mathrm{wait}}(i,16),$$

where b is a measure of the relative wait-times for one machine versus multiple machines and is empirically approximately 0.37 for NCSA machines and 0.19 for SDSC's IA-64. This derives from the data for wait-times: for NCSA, they are 530 and 2130 s, for SDSC they are 530 and 4200 s for 8 and 16 processors, respectively, while for NCSA+SDSC, the maximum wait-time is 780 s. Hence, overall, time-to-solution is

$$\begin{aligned} T_{\mathrm{s}}(\mathrm{NCSA},8;\mathrm{SDSC},8) &= T_{\mathrm{run}}(\mathrm{NCSA},8;\mathrm{SDSC},8) + T_{\mathrm{wait}}(\mathrm{NCSA},8;\mathrm{SDSC},8) \\ &\approx aT_{\mathrm{run}}(\mathrm{NCSA},16) + bT_{\mathrm{wait}}(\mathrm{NCSA},16) \\ &\approx 1.2\,T_{\mathrm{run}}(\mathrm{NCSA},16) + 0.37\,T_{\mathrm{wait}}(\mathrm{NCSA},16) \\ &= (T_{\mathrm{run}}(\mathrm{NCSA},16) + T_{\mathrm{wait}}(\mathrm{NCSA},16)) \\ &\quad + (0.2\,T_{\mathrm{run}}(\mathrm{NCSA},16) - 0.63\,T_{\mathrm{wait}}(\mathrm{NCSA},16)). \end{aligned}$$

For distribution to be beneficial (ideal case),

$$(0.2\,T_{\mathrm{run}}(\mathrm{NCSA},16) - 0.63\,T_{\mathrm{wait}}(\mathrm{NCSA},16)) \le 0$$

or

$$T_{\mathrm{run}}(\mathrm{NCSA},16) \le 3.15\,T_{\mathrm{wait}}(\mathrm{NCSA},16).$$

Thus, for all future jobs, with a predicted wait-time of T_{wait}(NCSA,16), we can safely distribute those jobs for which the expected T_{run}(NCSA,16) is $3.15\,T_{\mathrm{wait}}$(NCSA,16). However, in the practical world, we may have situations where

$$(0.2\,T_{\mathrm{run}}(\mathrm{NCSA},16) - 0.63\,T_{\mathrm{wait}}(\mathrm{NCSA},16)) > 0.$$

Thus, it is often a trade-off between acceptable wait-time versus additional CPU hours spent. It is important to point out that there are fluctuations due to changing work and queue loads in the values of the various parameters, e.g. a and b, and T_{wait}. However, the general results hold irrespective; specifically, the BQP result factors in these fluctuations in its probabilistic predictions.

5. Conclusion

In this paper, we have highlighted the need for and advantages of distributed job submission. We started from a simple, sequential MD code, parallelized it and extended it to run over distributed resources using MPICH-G2. Importantly, we showed that when running over multiple machines, even when simulating relatively small physical models, the performance is comparable with when running on a single machine. This formed the basis for the next phase of our work: investigating submission modes for a tightly coupled simulation in order to reduce T_{s}. Our approach was to circumvent the static model of fixing the number of processors on a predetermined set of resources using a co-scheduler; by contrast, we adopted an agile-execution model, where we determined the best resource and configuration to use almost at run-time.

We also have shown that, on average, the wait-time for a job submitted to a single machine can be longer than the wait-time of the same job when decomposed and submitted to multiple machines. We postulate that this is due to the typically lower resource requirement per machine. When combined with sophisticated tools such as the BQP, which provides a good estimate to a first approximation of the probability of a job to run in a given window, opportunistic decomposition has the clear potential to increase throughput—to lower wait-times significantly with relatively insignificant increase in CPU usage. We contend that this is the first documented work to use a prediction tool such as BQP at deployment (i.e. just before run-time) to decompose tightly coupled simulations *effectively*.

It is important to note that such opportunistic scheduling is not guaranteed to be successful; distributing over multiple machines will not always lower wait-times, as it is conceivable that the wait-time for any one of the small jobs on a machine could be larger than the wait-time for a single simulation on a larger machine, if not indefinitely long in the pathological case.

Although we show a reduction in T_{s} for a tightly coupled simulation decomposed over *multiple* machines, our approach is sufficiently general that we can decompose a large job into smaller jobs even on a *single* machine and still find a reduced T_{s}. Hence, the claim is that this is a novel submission mode. Also, it is worth mentioning that our approach is valid for ensembles of simulations, including high-throughput computing, where optimal aggregation strategies of 'small jobs' into 'big jobs' can be made. In some ways, the aggregation approach is the reverse of the decomposition approach, but it can be implemented across different machines. We will report results on this in a future paper.

This work is a part of the Cybertools project (http://www.cybertools.loni.org/) and is supported by NSF/EPSCoR award no. EPS-0701491. The authors would like to thank Wei Huang for help in generating large test models.

References

Chakraborty, P. & Jha, S. 2008 Design and performance analysis of a distributed HPC molecular dynamics code. In *MG '08: ACM SIGAPP 15th Mardi Gras Conf., Poster paper, Baton Rouge, LA, USA*. Washington, DC: ACM SIGAPP.

Manos, S., Mazzeo, M., Kenway, O., Karonis, N., Toonen, B. & Coveney, P. 2008 Distributed MPI cross-site run performance using MPIg. In *HPDC '08, Poster paper, Boston, MA, USA*, pp. 229–230. Washington, DC: ACM.