Database Assisted Distribution to Improve Fault Tolerance for Multiphysics Applications

Robert S. Pavel
Los Alamos National Laboratory
Los Alamos, NM 87545 USA
rspavel@lanl.gov

Allen L. McPherson
Los Alamos National Laboratory
Los Alamos, NM 87545 USA
mcpherson@lanl.gov

Timothy C. Germann
Los Alamos National Laboratory
Los Alamos, NM 87545 USA
tcg@lanl.gov

Christoph Junghans
Los Alamos National Laboratory
Los Alamos, NM 87545 USA
junghans@lanl.gov

ABSTRACT

Multiscale physics applications present an interesting problem from a computer science standpoint as task granularity has the potential to vary drastically which places a heavy burden upon the task scheduler and load balancer. Additionally, due to the long execution time of some of these computations, fault tolerance becomes a necessity as not being able to recover from a fault during a single long running task results in the recomputation of all data used to generate the inputs. Traditionally, this is facilitated through the use of checkpointing. However, these checkpoints must be taken sparingly due to their high cost.

In this paper, we describe our use of a NoSQL database and asynchronous task based runtimes to work directly from the checkpoints themselves with minimal code modifications by domain scientists. To evaluate the performance impact of this approach, we have studied the CoHMM proxy application: a co-design proxy application designed to test modern runtimes by simulating the propagation of a shock wave through a material through the use of the heterogeneous multiscale method. We distilled this proxy application to a library that we used to implement CoHMM in a range of runtimes with and without our database assisted approach and we measured the overhead of each with respect to the CoHMM application and the cost of serializing and migrating data in the runtimes themselves.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming, Distributed Programming*; D.4.5 [**Software**]: Operating Systems—*Reliability, Checkpoint/Restart*

General Terms

Performance, Reliability

Keywords

Co-Optimization for Multiple Objectives, Multiphysics, Runtimes, Database, Fault Tolerance, Resiliency

Co-HPC2015, November 15-20, 2015, Austin, TX, USA
©2015 ACM. ISBN 978-1-4503-3992-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2834899.2834908.

1. INTRODUCTION

Computer simulation of complex physical phenomena often requires the simultaneous execution and coordination of physics simulations spanning multiple time and length scales. This is achieved through the use of multiscale models that combine expensive small-scale simulations, such as a molecular dynamics simulation, with comparatively cheap larger scale simulations, such as finite volume methods. The combination of these lends themselves well to task and data parallelism and allow for considerably larger systems to be simulated.

However, as a result, the granularity of these tasks can be very large and, depending on the task granularity and the techniques used to improve performance, vary drastically, creating load imbalance. One way to resolve this is through the use of modern asynchronous task based runtime systems which have powerful load balancers that can schedule and migrate tasks in a way to avoid starvation and maintain throughput.

Unfortunately, even with the load imbalance issue resolved, fault tolerance remains. For long running scientific applications fault tolerance is already an important issue and this will only increase as we migrate from the petascale to the exascale era [12]. Traditionally this is handled through checkpointing [19] but, depending on task length, as well as the overall speed of the simulation, the frequency at which checkpoints are desired may be prohibitively high.

To that end, we propose an approach in which we simultaneously optimize for both performance and resilience by combining asynchronous task based runtimes with databases to provide a solution to these problems. By utilizing an asynchronous task based runtime, we are able to take advantage

of task scheduling and load balancers designed to handle a wide range of task granularities. However, rather than use the runtimes for data movement, we instead treat a database as a global memory. This allows us to largely avoid the serialization costs associated with distributed runtimes while allowing us to store checkpoints at a much higher frequency.

In this paper, we describe our approach and consider CoHMM; a proxy application developed by ExMatEx [1], a DOE/ASCR Co-Design Center focused on scale-bridging material science and engineering applications, designed to represent multiscale physics applications and simultaneously test modern runtime.

In this paper, we first provide brief background information in section 2. In section 3 we go into more detail on what motivated this avenue of research, and in section 4 we describe the CoHMM proxy application in greater detail. Section 5 explains our approach, and section 6 shows our experimental results and the overhead of this approach. In section 7 we discuss future improvements to our work and in section 8 we present our conclusions from this research.

2. BACKGROUND

In this section, we will provide a bit of background information on our work. First, we will provide a very brief overview of multiscale physics, the application of interest. Then we will discuss asynchronous task based runtimes with an emphasis on the specific runtimes of interest in this paper. Finally, we will briefly explain the rise of NoSQL databases and why they are of potential use to scientific computing.

2.1 Multiscale Physics Applications

Computer simulation of complex physical phenomena often requires the simultaneous execution and coordination of physics simulations spanning multiple time and length scales. This is because a fundamental challenge in scientific computing is reconciling the different scales at which different scientific models are applicable. For example, modeling the deformation of a plane's wing using a molecular dynamics simulation would be incredibly costly as molecular dynamics operates at the atomic level. Multiscale physics resolves this issue by utilizing simulations with multiple models, generally at different scales in time and/or space [16, 26].

The subset of interest for our work are multiscale physics applications in which a macroscale method dynamically spawns subscale model computations as needed when the constitutive response becomes too complex for the macroscale model. This allows phenomena at a very small scale to be simulated to determine essential properties of the material which can then be used, in conjunction with a coarser grained model and statistical properties, to advance the overall simulation [28].

2.2 Asynchronous Task Based Runtime

When tasked with writing such a complex multiscale physics code, developers typically draw on a set of languages and software tools such as C, C++, or FORTRAN, and employ MPI for communication. Effective use of acceleration devices (e.g. multi-cores or GPUs) is achieved using a specialized API such as CUDA [21], OpenCL [27], or OpenMP [14], leading to a combination colloquially termed "MPI+X".

This traditional approach has served the community well for many years, especially for single-physics, bulk synchronous codes. However it is becoming increasingly difficult to develop and maintain modern applications using traditional tools. Advanced functionality such as dynamic tasking, adaptive communication patterns, and fault tolerance must either be implemented directly by the application developer or specialized versions of the traditional tools must be employed.

This has led to a rise in programming models and runtimes designed with finer grain dependencies in mind. These runtimes implement models based on the scheduling of asynchronous tasks. Two such runtimes are Intel's Concurrent Collections and Charm++.

2.2.1 *Intel Concurrent Collections*

Intel Concurrent Collections (CnC) is a unified model for shared and distributed memory systems with an emphasis on providing tools for domain experts to express the algorithm with minimal scheduling constraints while allowing a tuning expert to later optimize the code [11]. CnC provides constructs to allow the programmer to decompose a C++ application into a set of asynchronous tasks with data dependencies that are then automatically distributed across the entire system with minimal input from the programmer. CnC relies upon the programmer for fault tolerance

2.2.2 *Charm++*

The Charm++ runtime and model was developed at the University of Illinois Urbana-Champaign and is a model built around medium-grained processes, called chares, that interact with one another via messages and callbacks [17]. Charm++ is designed with an emphasis on re-usability through modules and provides a high level language with which to express the dependencies between tasks and data. Newer releases of Charm++ provide checkpoint based fault tolerance [29]

2.3 NoSQL Databases

As the size of data centers increased, the limitations of, traditional, SQL [15] databases became apparent. One solution to this was the concept of the NoSQL [13] database which sacrifices many of the relational aspects of an SQL database for the simplicity of a key-value store.

Scalable, in-memory, NoSQL, databases [22] provide services that enable many new application capabilities with fault tolerance being an obvious example. An application, in collaboration with a scheduling and execution environment, can detect a component failure and restart the failed component from a previously cached data checkpoint. This enables recovery from a single node failure, as opposed to taking down the entire application - as often happens today.

These databases can also be used to accelerate codes by caching previously computed results to avoid expensive recomputation. Applications may also cache data for use by concurrently executing, in-situ, visualization and analysis code. Finally, databases can be used as a communication mechanism in place of messaging: components can store data into the database that is then easily retrieved by any other component within the system.

3. IMPROVING FAULT TOLERANCE OF APPLICATIONS

Fault tolerance is a growing concern as we approach the exascale era [12]. It is a particularly large concern in scien-

tific computing as runs can take hours, if not days, and a single fault can crash the system and potentially corrupt the data. Traditionally, checkpointing [19] is employed to allow programs to recover quickly from faults. However, the process of writing the current state to disc is costly and must be performed sparingly so as to not impact performance too heavily.

In previous work [23], we noticed the benefits of using a NoSQL database (Redis [9]) to store the results of previous costly operations and to act as a look up table to avoid recomputing values. In our testing, the overhead of the database was more than compensated for by the time we saved by minimizing the number of MD simulations. After continuing to work closely with domain experts, we investigated the benefits of storing all shared data in the database and relying on the runtime solely for task scheduling and load balancing. Instead of relying upon the runtime to migrate data we push all results to the database and merely transmit the minimum information required for the runtime to collect the results from the cloud.

In doing this, we treat the checkpoints as another level of memory that we are able to read and write to. This allows us to partially overlap the cost of saving the checkpoints with that of serializing the data to be transferred by the runtime itself. Furthermore, we push the task of ensuring availability and consistency of data to the databases themselves, where this is a heavily studied problem [13]. And, as a side benefit, we are able to rapidly implement our applications in a range of runtimes with minimal concerns for the underlying memory model or even programming language and use the runtimes solely for task scheduling.

Depending on performance and the benefit of such models, we can later make a full implementation that can better take advantage of the features of the runtime.

4. THE COHMM PROXY APPLICATION

The **Co**-Design **H**eterogeneous **M**ultiscale **M**odeling proxy application (CoHMM) is a proxy application [18] designed specifically with the goal of exercising modern runtime systems through the use of a multiscale physics application [1]. CoHMM implements a heterogeneous multiscale method [28] to simulate the propagation of a shockwave through a two dimensional material.

A full description of this is beyond the scope of this paper and is presented in previous works [24, 23], but a high level overview from a computer science perspective is as follows: At the coarsest scale, a "macrosolver" divides the system into small domains and handles the interactions between these domains, or "cells", through finite volume methods which are implemented as simple stencil operations. However, to advance this simulation requires the computation of the fluxes of each cell. This is handled with a much more costly **M**olecular **D**ynamics (MD) simulation, specifically the **Co**-Design **M**olecular **D**ynamics (CoMD) proxy application [1].

To minimize the cost of these MD simulations, we build upon previous work [23] and apply an interpolation scheme known as "kriging" [25] that utilizes previously computed values to potentially skip the expensive MD simulation. Furthermore, we store the results of every MD simulation in a database to avoid recomputing previously obtained results.

This proxy application was chosen as it is not only indicative of multiscale physics but it also provides a good variety of tasks. At the macrosolver level, the simple stencil performs a few floating point operations on a point and its nearest neighbors and is a very good candidate for tiling. For the flux computations, either a kriging task or an MD simulation is performed on each point. The MD simulation itself involves a single input point that is used to generate a system that is simulated for one thousand iterations, which is a time consuming operation. And, in between is the kriging task which involves database accesses, the solution of a small linear system of equations, and potentially an MD simulation on top of that if kriging fails. Thus, we have a comparatively simple small stencil operation followed by a large number of tasks of varying lengths and complexities, which provides a good test of the load balancers in the various runtimes.

5. METHODOLOGY AND IMPLEMENTATION

To this end, we studied CoHMM and converted it into a library with driver functions that divided each iteration of the macrosolver into a sequence of "steps", with each step spawning flux tasks. These tasks are then passed to the runtime to distribute the flux computations to the available processing units. To study the overhead of our approach, we developed two implementations for each runtime. Each implementation uses the same underlying library functions to advance the simulation and differ only in how tasks are queued and how input and output data are handled.

By requiring domain scientists to write their initial code with a focus on having steps with clear inputs and outputs we are easily able to convert the code into a functional library. The drivers for these libraries can be rapidly implemented for a variety of runtimes and distribution schemes with no additional modifications by the domain scientist.

This approach, while less efficient and closer to a bulk synchronous approach, avoids the need to re-implement the scientific library for each runtime and instead treats the scientific library as a black box the runtime interacts with. This allows for a wider range of programming models to utilize the same library as even the programming language becomes less important so long as the library can be called. Furthermore, by employing asynchronous task based runtimes we are better able to take advantage of task migration and are able to rely on the runtime to handle the load balancing of the different task lengths.

The implementation of this work may be downloaded from the CoHMM proxy application's page on Github [2]. The traditional implementation can be found under the "Bold and Distributed (bad)" branch, with the database assisted version found under the "Database Assisted Distribution (dad)" branch.

5.1 Traditional Runtime Implementation

The first implementation is a more traditional implementation in which the runtime handles all task and data migration. As macrosolver steps generate tasks, they are enqueued, with input data, into the runtime's queue. The runtime then proceeds to schedule the execution of the task, collect the results, and return them for the next macrosolver step.

Due to the comparatively small size of the problem at the macrosolver level, the macrosolver itself is a single task

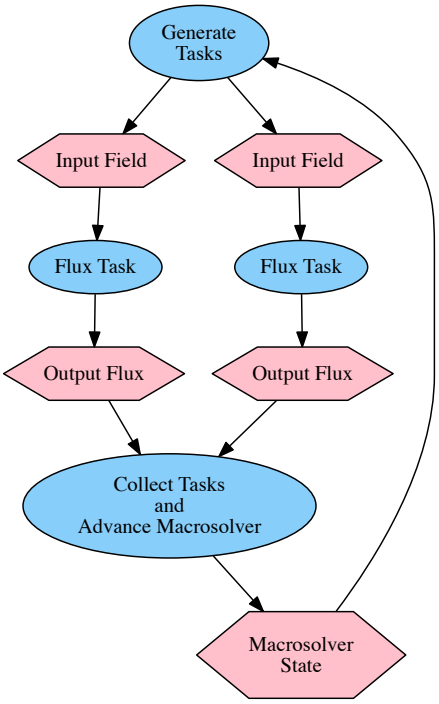

Figure 1: Simplified Task Graph of Traditional Implementation

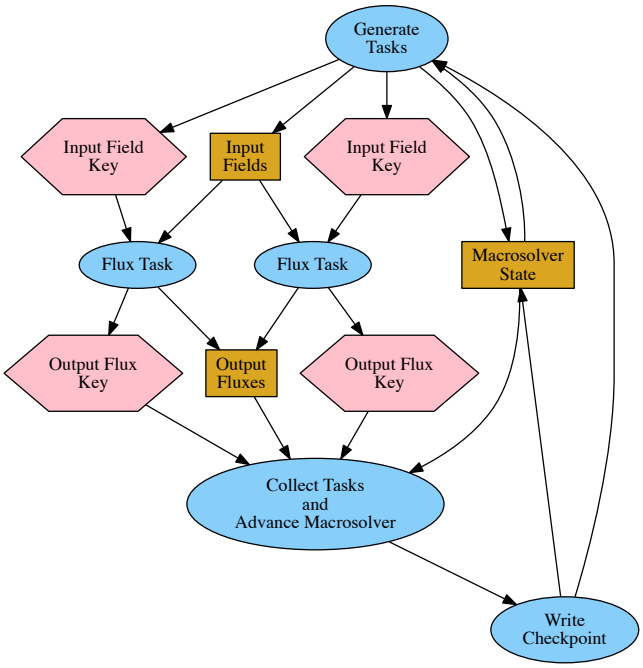

Figure 2: Simplified Task Graph of Database Assisted Distribution

without tiling and the state of the macrosolver is kept in memory on the root process. For the sizes of the problems being studied, the costs of data migration and ghost cell exchanges largely outweigh the increased throughput.

A simplified task graph is shown in Figure 1 with pink hexagons representing arguments. Each Flux Task uses an Input Field to generate an Output Flux, and the results are used to advance the Macrosolver State. In actuality, fluxes are used to adavance the macrosolver state four times per time step.

Additionally, while not used to share data between processes within an interation, a database is used to support kriging and to avoid unnecessarily recomputing previously obtained data. Details of the behavior of this can be found in previous work [23]. For the purposes of our work it is simply a technique used to drastically accelerate the overall simulation.

5.2 Database Assist Distribution

The second implementation is the cloud based solution. As tasks are generated by macrosolver steps, their associated data is put to the database. Upon completion of a step, the number of tasks is passed to the runtime. The runtime then generates the specified number of skeleton tasks and schedules them across the available processing units. Each task then reads the required inputs from the database, computes the flux for the cell, and writes the result to the database. Once all tasks have signaled completion, the root processor begins the next macrosolver step.

As before, the macrosolver itself is a single task without tiling. However, in this implementation the macrosolver also writes its current state to the database at the end of each step. Upon starting the next step, it reads its state, as well as the result of the flux tasks, to advance the overall simulation.

The simplified task graph of this version can be seen in Figure 2. While similar in layout to Figure 1, only the keys to the database (yellow rectangles) are passed to and from tasks. Instead, each task uses the provided key, and the overall program state, to read and write results to the database.

In addition to being used to share data between processes, the database is once again used to support kriging and to avoid unnecessarily recomputing previously obtained data.

The database assisted version also checks if a macrosolver iteration has already completed and, if so, relies on the existing data in the database. If a fault occurs mid-iteration, we rely on CoHMM's normal database checks to treat the previously completed flux calls as reads from a look-up table.

All benchmarks in this paper are based upon a centralized database. While we leave the issue of configuration and ensuring consistency to the user as it is very much dependent upon the host system, we provide optional interfaces that are detailed in our documentation and source code [2].

6. EXPERIMENTS AND RESULTS

While our goal is to simultaneously optimize for both performance and resiliency, our database assisted approach is inherently going to result in a performance penalty, if only because tasks will begin without all data available. Depending on the application, this may or may not be an acceptable price to pay for the greatly increased resiliency due to fault tolerance. To make this decision, it is important to understand how much overhead is incurred.

To measure this, we compared the execution times of the traditional implementation, in which the runtime handles data migration, as well as our database assisted implementation. Both used a database for the purpose of kriging, but

only the database assisted implementation used the database as a global shared memory.

For the purposes of this paper, we focused on two runtimes: Intel's Concurrent Collections [11] (CnC) and Charm++ [17]. CnC and Charm++ were chosen as both are representative of modern asynchronous task based runtimes and both have demonstrated high performance and scalability in the past. This allows us to focus more on the overhead and scalability of our approach as opposed to that of the underlying runtime.

Similarly, we utilized a single Redis [9] server as our database. Our approach supports distributed databases but the performance of these depends upon how the database itself is distributed. As our focus is on our database assisted distribution and less on the databases themselves, we chose to utilize a centralized database with proven scalability for the purpose of these benchmarks and avoided systems where a more distributed database would be required. For obvious reasons, this limits the resiliency of our benchmarks and production runs should determine the appropriate configuration for a distributed database.

All benchmarks were run on the Los Alamos National Laboratory's Conejo supercomputer in which each node consists of two quad-core Intel Xeon X5550 processors for a total of 8 physical processor cores per node. Both implementations interact with version 3.0.2 of the Redis database with version 0.13.1 of the hiredis library [10] and use Intel's Math Kernel Library version 11.2 [5] during the kriging process. The CnC implementations were compiled with Intel's C++ Compiler version 15.0.090 [4], Intel MPI version 5.0.1 [6], and Intel CnC version 1.0.002 [6]. The Charm++ implementations were compiled with the GNU C++ Compiler version 4.8.2 [3], Open MPI version 1.6.5 [7], and Charm++ version 6.6.0.1 [8].

In terms of the problem size, we chose to focus on strong scaling results and fixed the problem as a centralized shock discontinuity in a 128×128 grid and twenty iterations of the macrosolver. This problem was selected as the number of required flux operations per phase is sufficiently high to ensure that we are able to see the impact of the runtimes' load balancers even at 512 processors.

6.1 CoHMM with CoMD

First we evaluated the overhead with the full proxy application, including CoMD calls with a measured execution time of approximately 8 seconds per call. We compared the traditional implementation (RT) with the database assisted distribution (DB) and plotted the results in Figure 3. Due to the size of the problem and the execution time of the simulation, we were able to collect data for as few as 2 nodes (16 processors) with CnC and 4 nodes (32 processors) for Charm++.

For both the CnC and Charm++ implementations, the version that utilizes the DB has consistently lower performance. This was to be expected as each task must now request and wait for data during execution and can't rely on the runtime to prefetch the inputs. Fortunately, the scalability does not differ between the RT and DB implementations. We show this by plotting the execution time relative to the run with the smallest number of processors (so 16 for CnC and 32 for Charm++) in Figure 4.

In both cases, speed-up is near linear. with the CnC implementation even having near perfect speed-up out to 256

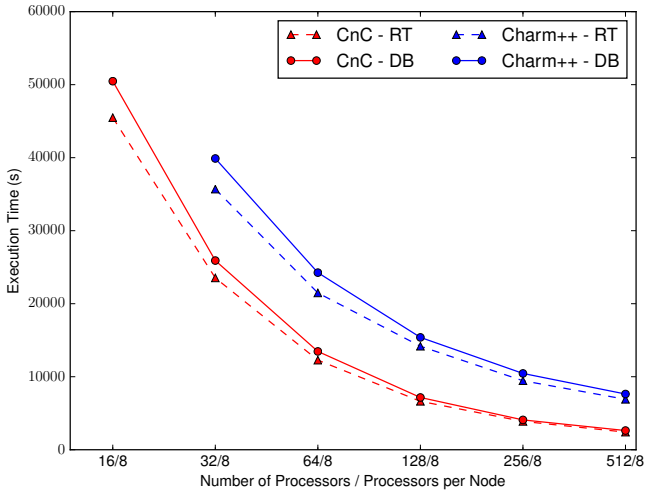

Figure 3: Performance of CoHMM with CoMD Call

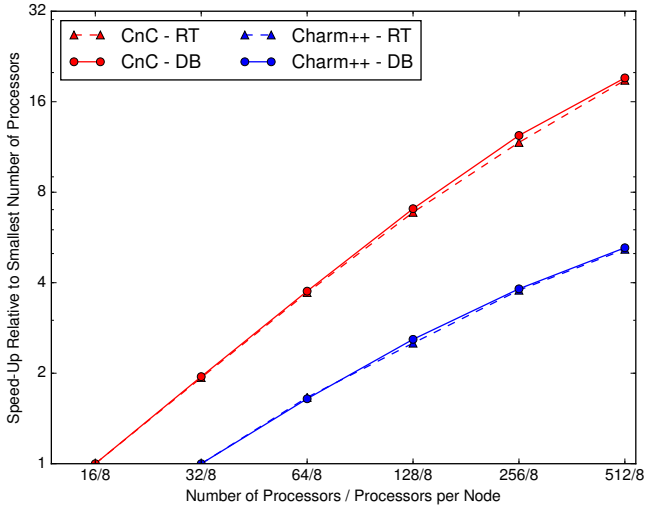

Figure 4: Plot of Speed-Up of CoHMM with CoMD Call

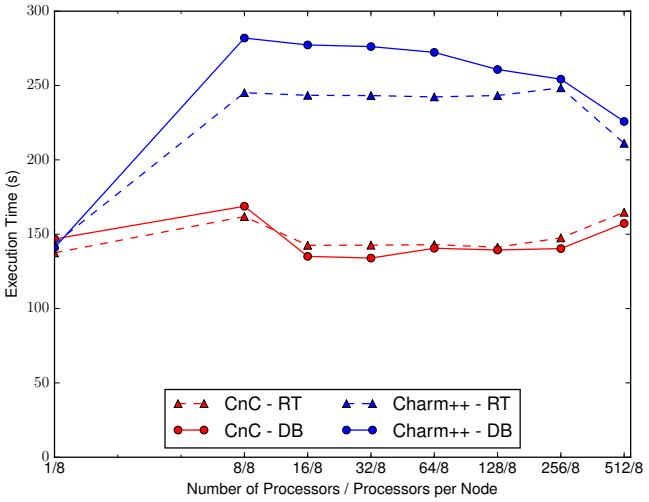

Figure 5: Performance of CoHMM with Analytic Solution

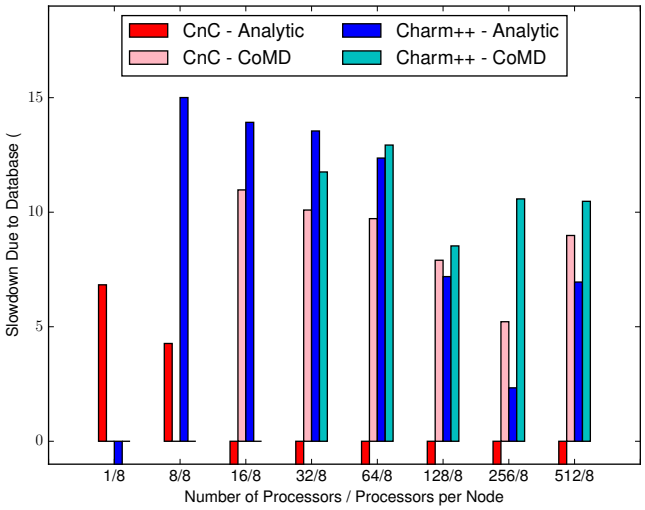

Figure 6: Overhead of Database Assisted Distribution

processors (32 nodes of Conejo). This is important as it shows that the overhead of our approach is not sufficient to impact the scalability of the application and traditional optimizations will still apply. The Charm++ data in particular is valuable as it shows that this is true both under linear scaling, up to 64 processors (8 nodes of Conejo) as well as when the point of diminishing returns is reached.

6.2 CoHMM with Analytic Solution

Next, we disabled CoMD and instead used an analytic solution to compute the fluxes. The approximate execution time of a CoMD call is 8 seconds, whereas the approximate execution time of the analytic solution was measured as 0.002 seconds. As such, the execution time is primarily the overhead of the runtime and database accesses themselves with respect to the communication patterns of CoHMM. The execution times are plotted in Figure 5.

In both cases, the runtime and database overheads are fairly constant out to 512 processors, which is consistent with the scaling shown in Figure 4. These results are particularly interesting as, for CnC, the runtime version is actually outperformed by the database version when more than one node of the cluster is in use. This appears to be due, in part, to the CnC runtime's serialization framework. Additionally, as this only occurs once inter-node communication begins, it also likely has to do with how CnC handles data migration between nodes of a system. Under the Intel CnC runtime, if a task begins execution and determines input data is not available it will terminate execution and restart the task from the start at a later time [6]. This is in contrast to the DB version which simply blocks.

Comparatively, the Charm++ version exhibits almost the opposite behavior. The single threaded execution has the database version outperforming the runtime version, likely due to unnecessary serialization of data between tasks running in the same shared address space, But expected behavior resumes as of the multithreaded (8 processors on a single node) configuration.

This data is valuable as it shows that the cost of sharing data through the runtime may actually be greater than sharing it via a database. With more intelligent pre-fetching, or even integration of the database into the runtime, this may improve performance overall.

6.3 Overhead of Database Assisted Distribution

Finally, we measured the overhead of our approach by comparing the execution times for the aforementioned experiments and measuring the slowdown of our implementation versus the traditional runtime implementation. The results of this can be seen in Figure 6.

In all cases, the overhead compared to the traditinial implementation decreases as the number of processor nodes increases. While counterintuitive, this is due to the underlying runtime load balancers being better suited toward taking advantage of shared memory as well as knowing what tasks are assigned to what processor in advance. As the number of processor nodes increases, a larger percentage of time is spent transferring data through distributed memory and these features become less advantangeous.

While not negligible, the average overhead for the Charm++ implementation, with full CoMD calls, is 10.85%. For CnC it is only 8.82%. While this may seem like a large penalty, it may be an acceptable price to pay if the risk of a fault is sufficiently high.

A negative overhead corresponds to a case in which the traditional runtime implementation was outperformed by the database assisted implementation.

7. FUTURE WORK

Currently, we focused on comparatively small problem sizes so as to focus more on the approach than the implementation. As such, our CoHMM implementation is limited to a single level of parallelism: flux computation tasks. However, for larger problem sizes we hope to extend our approach to exhibit more hierarchical parallelism by tiling at the macro-solver level in addition to spawning tasks to compute the flux computations.

Additionally, our current approach relies upon a NoSQL database. While key–value stores meet our current needs, we hope to adapt this to more traditional databases as well

as consider ways to take advantage of other technologies, such as burst buffers [20].

We currently rely on the database to ensure the atomicity of all writes. This may not remain viable and we may need to implement ways to verify that the data is not corrupted.

Additionally, our results with the analytic solution in section 6.2 suggest that, under certain circumstances, writing and reading from an in-memory database can be less costly than utilizing the serialization and data migration frameworks built in to many runtimes. We wish to further investigate this and possibly make a more tightly coupled approach that builds upon this.

8. CONCLUSIONS

As the migration toward larger and more powerful computers continues, fault tolerance becomes an even bigger concern. This is incredibly important for scientific computing where runs can take hours, if not days, and a single fault can crash the system and potentially corrupt the data. While checkpointing is a solution, the overhead involved limits the frequency at which checkpoints can be made.

In this paper, we propose a solution to this problem for multiphysics applications where the desired checkpointing frequency is high. We combine asynchronous task based runtimes with in-memory databases to take advantage of the task scheduling and load balancing of the runtimes as well as a database assisted distribution in which we effectively work directly from the checkpoints themselves. Instead of using the runtimes for data migration we read and write to the database with the runtime solely responsible for scheduling work. In the event of a fault, we are able to instantly resume as the complete state of the system is already preserved. This also allows us to more effectively overlap the serialization costs associated with distributed runtimes with the cost of reading and writing to the database itself.

We studied this by focusing on the CoHMM proxy application which was designed to be representative of multiscale physics applications while simultaneously testing modern runtimes. We reduced the CoHMM proxy application to a sequence of library calls and we implemented the application both with a more traditional approach in which the runtime handled all data migration as well as our database assisted version. By studying the performance of this approach, we have determined that it is potentially beneficial for multiphysics applications. While there is a performance penalty, it is, on average, 8.82% with CnC and 10.85% with Charm++. Depending on the likelihood of a fault and the execution time of a given task, this can be an acceptable performance penalty.

Additionally, with a greater focus on the database distribution and a more hierarchical approach these costs may decrease even more. This is suggested by our results with the analytic solution which show that, under certain circumstances, the in-memory database may be less costly than the serialization and data migration frameworks of the runtimes themselves.

Furthermore, in addition to greatly increasing the fault tolerance of scientific applications this approach has the benefit of allowing for the rapid implementation of applications in multiple runtime models. And, while the performance will not be anywhere near as high as if we were to take advantage of the more fine grain dependencies allowed by many of these asynchronous task based models, this does give an idea of the performance costs of the database assisted distribution as well as a comparatively simple path by which existing scientific codes can begin to take advantage of these runtimes with comparatively minimal modifications.

9. ACKNOWLEDGMENTS

This work was supported by the Los Alamos Information Science and Technology (IS&T) Co-Design Summer School and the U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research (ASCR) through the Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx), and the Center for Nonlinear Studies (CNLS).

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Los Alamos National Security, LLC for the National Nuclear Security Administration (NNSA) of the U.S. DOE under contract DE–AC52–06NA25396. The authors would also like to acknowledge the Institutional Computing Program at LANL for use of their HPC cluster resources. Additional simulations were performed on the CCS-7 cluster Darwin.

This work was approved for unlimited release under LA-UR-15-26622.

10. REFERENCES

[1] *ExMatEx: Extreme Materials at Extreme Scale.* http://www.exmatex.org/.

[2] exmatex/CoHMM.

[3] GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF). `https://gcc.gnu.org/`.

[4] Intel® C++ Compilers | Intel® Developer Zone.

[5] Intel® Math Kernel Library (Intel® MKL) | Intel® Developer Zone. `https://software.intel.com/en-us/intel-mkl`.

[6] Intel® MPI Library | Intel® Developer Zone. `https://software.intel.com/en-us/intel-mpi-library`.

[7] Open MPI: Open Source High Performance Computing. `http://www.open-mpi.org/`.

[8] Parallel Programming Laboratory | UIUC. `http://charm.cs.uiuc.edu/software`.

[9] *Redis.* http://redis.io/.

[10] redis/hiredis. `https://github.com/redis/hiredis`.

[11] Z. BudimliÄĞ, M. Burke, V. CavÃl', K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and others. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[12] F. Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[13] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[14] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[15] C. J. Date and H. Darwen. *A guide to the SQL Standard: a user's guide to the standard relational*

language SQL, volume 55822. Addison-Wesley Longman, 1993.

[16] D. Givon, R. Kupferman, and A. Stuart. Extracting macroscopic dynamics: model problems and algorithms. *Nonlinearity*, 17(6):R55, 2004.

[17] L. V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, Washington, D.C., USA, 1993. ACM.

[18] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, and others. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013), Boston, USA*, 2013.

[19] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan. 1987.

[20] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, Apr. 2012.

[21] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. June 2011.

[22] E. Redmond and J. Wilson. *Seven Databases in Seven Weeks*. Pragmatic Bookshelf; O'Reilly, 2012.

[23] D. Roehm, R. S. Pavel, K. Barros, B. Rouet-Leduc, A. L. McPherson, T. C. Germann, and C. Junghans. Distributed Database Kriging for Adaptive Sampling (D2kas). *Computer Physics Communications*, 192:138–147, 2015.

[24] B. Rouet-Leduc, K. Barros, E. Cieren, V. Elango, C. Junghans, T. Lookman, J. Mohd-Yusof, R. S. Pavel, A. Y. Rivera, D. Roehm, and others. Spatial adaptive sampling in multiscale simulation. *Computer Physics Communications*, 185(7):1857–1864, 2014.

[25] M. L. Stein. *Interpolation of spatial data: some theory for kriging*. Springer, 1999.

[26] M. O. Steinhauser. *Computational multiscale modeling of fluids and solids: theory and applications*. Springer, Berlin ; New York, 2008.

[27] The Khronos Group. *OpenCL âĂŞ The open standard for parallel programming of heterogeneous systems*. 2014. http://www.khronos.org/opencl/.

[28] E. Weinan, B. Engquist, X. Li, W. Ren, and E. Vanden-Eijnden. Heterogeneous multiscale methods: a review. *Commun. Comput. Phys*, 2(3):367–450, 2007.

[29] G. Zheng, L. Shi, and L. V. KalÃl'. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Cluster Computing, 2004 IEEE International Conference on*, pages 93–103. IEEE, 2004.