Tabulated equations of state with a many-tasking execution model

Matthew Anderson[1], Maciej Brodowicz[1], Hartmut Kaiser[2],
Bryce Adelstein-Lelbach[2], Thomas Sterling[1]

1 Center for Research in Extreme Scale Technologies
Indiana University
Bloomington, Indiana, USA
2 Center for Computation and Technology
Louisiana State University
Baton Rouge, LA, USA

ABSTRACT

The addition of nuclear and neutrino physics to general relativistic fluid codes allows for a more realistic description of hot nuclear matter in neutron star and black hole systems. This additional microphysics requires that each processor have access to large tables of data, such as equations of state, and in large simulations, the memory required to store these tables locally can become excessive unless an alternative execution model is used. In this work we present relativistic fluid evolutions of a neutron star obtained using a message driven multi-threaded execution model known as ParalleX. The goal of this work is to reduce the negative performance impact of distributing the tables. We introduce a component based on the notion of a "future", or nonblocking encapsulated delayed computation, for accessing large tables of data, including out-of-core sized tables. The proposed technique does not impose substantial memory overhead and can hide increased network latency.

Keywords

Astrophysics applications, ParalleX, HPX, Futures

1. INTRODUCTION

Future achievements in leading-edge science demand innovations in parallel computing models and methods to improve efficiency and dramatically increase scalability. One controversial issue is the relative value of global address space models and management versus more conventional distributed memory structures. This paper demonstrates one important use of global address space in the context of the ParalleX many-tasking execution model in order to enable simulation improvements in the domain of astrophysics that are not feasible using conventional practices.

Accurate treatment of hot nuclear matter in astrophysical compact object simulation is becoming increasingly important in the search for coincident detection of gravitational radiation and electromagnetic or neutrino emissions originating from the same source. Recent simulations have shown that the gravitational wave signature from a binary neutron star merger or a neutron star–black hole merger may even reveal important empirical details about the neutron star equation of state itself [27, 22].

The addition of nuclear and neutrino physics to general relativistic fluid codes allows for a more realistic description of hot nuclear matter in neutron star and black hole systems. Unfortunately, most of these and other microphysics routines cannot be computed in place; they must be precomputed in a large table which is then read and interpolated as the relativistic hydrodynamics simulation proceeds. Accurate neutron star simulations will increasingly rely upon ever larger tables of microphysics data that must be stored into memory, searched, and interpolated [25]. As the memory requirements grow for these tables, approaching and often exceeding the size of the physical memory of a single locality, the need to reduce the negative impact of distributing the tables becomes increasingly important. That is the precisely the goal of this work.

In this work we explore the experimental execution model called ParalleX [16, 20, 33, 13] as a means of addressing the critical computational requirement of dealing with very large tables containing microphysics. Adaptive task based approaches may provide other performance benefits to a distributed relativistic hydrodynamics simulation, but the focus of this work will be on reducing the negative impact of distributing the tables, while introducing minimal changes to the application code.

ParalleX is a synthesis of complementing semantic constructs delivering a dynamic adaptive framework for message-driven multi-threaded computing in a global address space context with constraint-based synchronization to exploit locality and manage asynchrony. The result is introspective runtime alignment of computing requirements and computing resources while permitting asynchronous operation across physically distributed resources. ParalleX was first implemented in the form of the HPX runtime system [32, 4]. HPX was developed to support the semantics and mechanisms comprising ParalleX targeting conventional SMP and commodity cluster computing platforms. As such, HPX is an experimental software package which not only tests the semantics of ParalleX but also measures the overhead costs of a software implementation and provides a prototype of a next generation runtime system for extreme scale applications.

The outline of this paper as follows: Section 2 gives an overview of the work related to this effort; related to Section 3 describes the relativistic fluid evolution, initial data, numerical methods, and equation of state details; Section 4 gives a brief overview of the HPX runtime system implementation of ParalleX; Section 5 describes how the equation of state is distributed across localities and how “futures”, or nonblocking encapsulated delayed computations, are used to hide network latency in table access; Section 6 gives neutron star evolution performance results using the Shen equation of state comparing the futures approach of accessing the equation of state table with that of reading in the table for every core (referred to hereafter as the conventional approach); Section 7 gives our conclusions and implications for future work.

2. RELATED WORK

Combining a many-tasking execution model implementation with a global address space for use in multiphysics has been discussed several times in the literature but experimental results are rare. There are multiple many-tasking runtime systems and libraries available for experimentation: Charm++ [21], Unified Parallel C (UPC) [8], Intel Threading Building Blocks library (TBB) [28, 23], HPX, Cilk plus [7], Chapel [9], Qthreads [34], the SWift Adaptive Runtime Machine (SWARM) [3], X10 [10], and even OpenMP [14] with some limitations. Among these many-tasking execution models and libraries, a small subset supports a global address space model such as Partitioned Global Address Space (PGAS) or Active Global Address Space (AGAS). This latter group consists of Charm++, UPC, X10, and HPX. Prior work in Charm++ describes a scheme for using shared arrays in multiphysics simulations along with a verification strategy for error detection [24] but does not present performance or implementation results. The proposed Phasers concept in X10 would also enable incorporation of equations of state tables in multiphysics simulations [31] but such an implementation has not yet been reported. An OpenMP based implementation is presented later in this work in order to contrast with the HPX implementation; in the physics literature, OpenMP is the preferred tool for this type of task [26].

3. RELATIVISTIC HYDRODYNAMICS

The work presented here adopts the flux-conservative formulation of the relativistic magnetohydrodynamics equations presented in [5] and includes high-resolution shock capturing (HRSC) methods. To calculate the numerical fluxes, we use the Piecewise Parabolic Method (PPM) [11] for reconstructing fluid variables. The approximate Riemann solver employed is Harten-Lax-van Leer-Einfeldt (HLLE) [17]. While the code is capable of also evolving magnetic fields, no magnetic fields were added to the initial data at this stage. Neutron star simulations presented in Section 6 were conducted using the Cowling approximation.

The tabulated equation of state used for generating the initial neutron star and all subsequent fluid evolution is the Shen equation of state [29]. A tabulated Shen equation of state was provided by C. Ott and is available for download at [2]. The publicly available tabulated Shen equation of state used for tests here is 288 MB; however, an updated Shen equation of state has since been released [30], and we have created a table based on this update which is 5.9 GB in size. Consequently, results from both tables will be presented in the results section. The neutron star initial data was generated using the Lorene libraries [1]; the neutron star has a mass of 1.4 solar masses and radius of 15.947 km. Tables related to neutrino transport, including neutrino opacity, were not included in this work but are part of future work.

The entire relativistic magnetohydrodynamics solver has been implemented in C++ using the HPX runtime system for parallelism. The application has been modularized for future incorporation into the HPX Adaptive Mesh Refinement (AMR) toolkit, although simulations for this work performed all computations with a uniform grid. The next section will give a brief overview of HPX and introduce the concepts crucial for asynchrony management in tabular access and parallel computation in general.

4. THE HPX RUNTIME SYSTEM

The C++ prototype runtime implementation of ParalleX is called High Performance ParalleX (HPX). A walkthrough description of the HPX architecture is found in Figure 1. An incoming parcel (delivered over the interconnect) is received by the parcel port. One or more parcel handlers are connected to a single parcel port, optionally allowing to distinguish different parts of the system as the parcel’s final destination. Here, locality is a ParalleX term identifying synchronous domain of computation, such as a single compute node in a cluster. The main task of the parcel handler is to buffer incoming parcels for the action manager. The action manager decodes the parcel, which contains an action bundled with relevant operands. An action is either a global function call or a method call on a globally addressable object. The action manager creates a HPX-thread based on the encoded information.

All HPX-threads are managed by the thread manager, which schedules their execution on one of the OS-threads allocated to it. HPX threads are implemented as user level threads, which decreases the costs associated with their creation, destruction, and state updates by minimizing the number of interactions with the OS kernel. HPX creates one worker OS-thread for each available core, whose purpose is to carry out the majority of computations in an application. Several scheduling policies have been implemented for the thread manager, such as the global queue scheduler, where all cores pull their work from a single global queue, or the local queue scheduler, where each core pulls its work from a separate queue. The latter supports work stealing for better load balancing. In the local scheduler, a queue is created for each of the worker OS-threads. When a worker thread is searching for work, it first checks its own queue. If there is no work there, the OS-thread begins to steal work by searching for work in other queues, first from its own non-uniform memory access (NUMA) domain, then from cores located on different NUMA domains.

If a possibly remote action has to be executed by an HPX-thread, the action manager queries the active global address space (AGAS) to determine whether the target of the action is local or remote to the locality that the HPX-thread is running on. If the target happens to be local, a new HPX-thread is created immediately and passed to the thread manager. This thread encapsulates the work (function) and the corresponding arguments for that action. If the target is remote, the action manager creates a parcel encoding the action (i.e. the function and its arguments). This parcel is handed to the parcel handler, which makes sure that it gets sent over the interconnect, causing a new HPX-thread to be created at the target locality.

The Active Global Address Space (AGAS) provides global

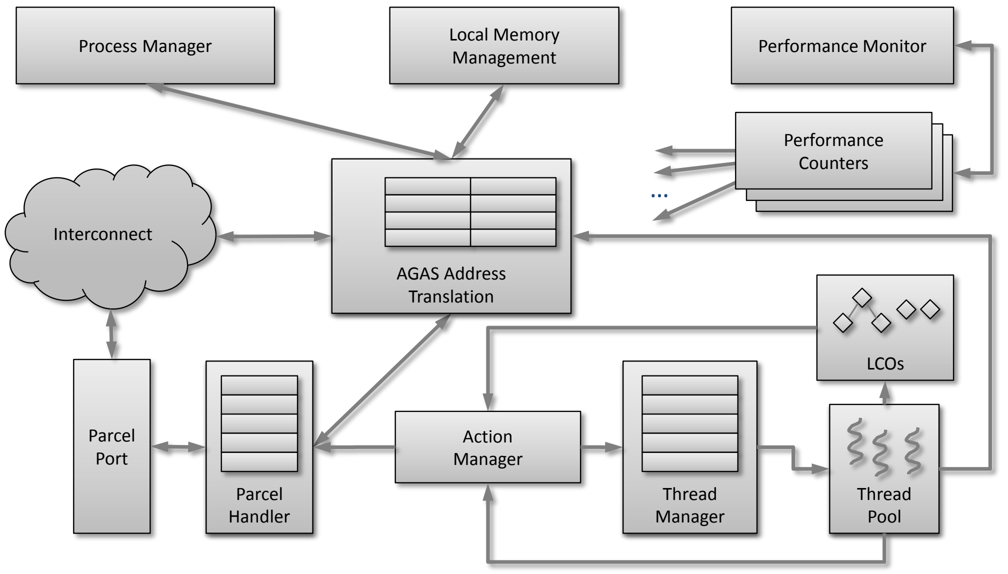

Figure 1: Modular structure of HPX implementation. HPX implements the supporting functionality for most of the elements needed for the ParalleX model: AGAS (active global address space), parcel port and parcel handlers, HPX-threads and thread manager, ParalleX processes, LCOs (local control objects), performance counters enabling dynamic and intrinsic system and load estimates, and the means of integrating application specific components.

address resolution services that are used by the parcel port and the action manager. AGAS addresses are 128bit unique global identifiers (GIDs). AGAS maps these global identifiers to local addresses and additionally provides symbolic mappings from strings to GIDs. The local addresses to which the GIDs are bound are typed, providing a degree of protection from type errors. Any object that has been registered with a GID in AGAS is addressable from all localities in an instance of the HPX runtime. AGAS also provides a powerful reference counting system which implements transparent and automatic global garbage collection.

Lightweight Control Objects (LCOs) are the synchronization primitives upon which HPX applications are built. LCOs provide a means of controlling parallelization and synchronization of HPX-threads. Semaphores, mutexes and condition variables [12] are all available in HPX as LCOs. Futures [6] are another type of LCO provided by HPX and are discussed in greater detail later in this paper.

Local memory management, performance counters (a generic monitoring framework), LCOs and AGAS are all implemented on top of an underlying component framework. Components are the main building blocks of remotely executable actions and can encapsulate arbitrary, possibly application specific functionality. Actions are special types which expose the functionality of a (possibly remote) function. An action can be invoked on a component instance using only its GID, which allows any locality to invoke the exposed methods of a component. In the case of the aforementioned components, the HPX runtime system implements its own functionality in terms of this component framework. Typically, any application written using HPX extends the set of existing components based on its requirements.

The relativistic hydrodynamics simulations make use of all the key features of HPX. The most crucial feature for contention and network latency hiding in tabulated equation of state access is the Future [6, 15]. The next section will describe the strategy adopted in HPX for distributing a large table and hiding network latency.

5. USING THE SHEN EQUATION OF STATE TABLES

The Shen equation of state (EOS) tables of nuclear matter at finite temperature and density with various electron fractions within the relativistic mean field (RMF) theory are a set of three dimensional data arrays enabling high precision interpolation of 19 relevant parameters required for neutron star simulations. As noted in Section 3, the publicly available Shen equation of state table is relatively small in size (288 MB); however, the most recent Shen table created for the neutron star evolutions presented here is 5.9 GB in size. Results using both tables will be presented in Section 6. In the case of the larger table, loading the whole data set into main memory on each locality is not feasible.

5.1 Interpolation Technique and Characteristics

The values of each of 19 variables describing the Shen EOS are contained in individual 3-D tables stored in memory in a row-major fashion. Single table data are arranged as samples of a single physical quantity computed at coordinates laying on a regularly spaced grid. The sizes of each grid vary from $220 \times 180 \times 50$ for the smaller 288 MB set to $440 \times 360 \times 130$ for the large, 5.9 GB set. The dimensions correspond to baryon mass density, temperature, and electron fraction, respectively.

To obtain the value of a variable at an arbitrary point within the 3-D domain, a $2 \times 2 \times 2$ cube of double-precision floating numbers must be accessed. The result is computed using a simple tri-linear interpolation from these values. Our neutron star simulations require only 8 out 19 tabulated quantities, which helps reduce the memory pressure. How-

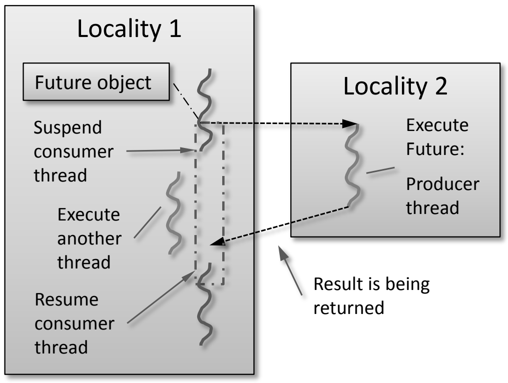

Figure 2: Schematic of a Future execution. At the point of creation of the Future, its encapsulated execution is started. The consumer thread is suspended only if the result of executing the Future has not returned yet. In this case the core is free to execute some other work (here 'another thread') in the meantime. If the result is available, the consumer thread continues without interruption to complete execution. The producer thread may be executed on the same locality as the consumer thread or on a different locality, depending on whether the target data is local or not.

ever, even when using a single instance of the smaller set of tables on each node to permit the efficient sharing of table data among all cores, the aggregate size of accessed data volume still significantly exceeds the combined size of L3 processor caches. Given that, in each interpolation request, effectively at most $2 \times 8 = 16$ out of 64 bytes per cache line (x86 architecture) are used and the coordinate stream generated by the application is random, the performance of interpolation function is memory bound.

5.2 The Overhead of Futures

Many HPX applications, including the relativistic hydrodynamics simulation detailed here, utilize Futures for ease of parallelization and synchronization. For this reason, the overheads of management and access of these constructs are a large factor in the total overhead of the HPX runtime in our code. In this subsection, we give a description of Futures, outline a performance test for measuring the overhead of Futures, and present the results of the test.

As shown in Figure 2, a Future encapsulates a delayed computation. It acts as a proxy for a result initially not known, most of the time because the computation of the result has not completed yet. The Future synchronizes the access of this value by optionally suspending HPX-threads requesting the result until the value is available. When a Future is created, it spawns a new HPX-thread (either remotely with a parcel or locally by placing it into the thread queue) which, when run, will execute the action associated with the Future. The arguments of the action are bound when the Future is created. Once the action has finished executing, a write operation is performed on the Future. The write operation marks the Future as completed, and optionally stores data returned by the action.

When the result of the delayed computation is needed, a read operation is performed on the Future. If the Future's action hasn't completed when a read operation is performed on it, the reader HPX-thread is suspended until the Future is ready.

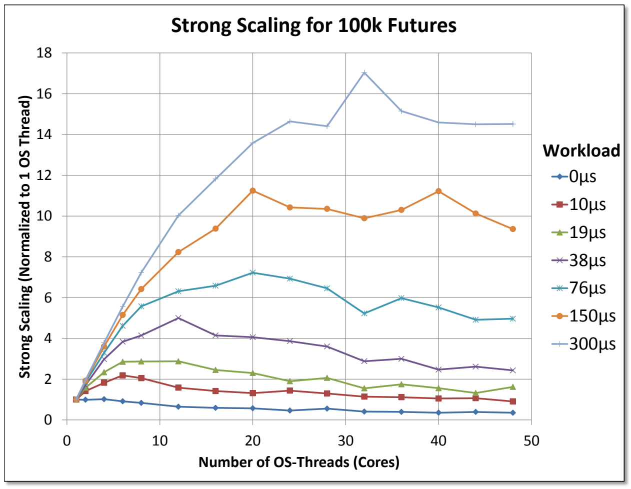

Figure 3: Results of the Future overhead benchmark. In each test, 100,000 Futures were invoked, with varying workloads. Each data set shows the strong scaling results for a particular workload.

Our benchmark for Future overhead created a fixed number of Futures, each of which had a fixed workload. Then asynchronous read operations were performed on the Futures until all of the Futures had completed. A high resolution timer measured the wall-time of the aforementioned operations. The test was run on an 8-socket HP ProLiant DL785 (each socket sports a 6-core AMD Opteron 8431) with 96 GB of RAM (533 MHz DDR2). Varying workload sizes and OS-thread counts were used. Five runs were performed for each combination of the parameters and the results were averaged to produce a final dataset. The numbers are presented in Figure 3.

On the locality we used for this benchmark, the amortized overhead of a Future is approximately *17 microseconds*. This overhead includes the time required to create the Future instance, start the evaluator thread, perform synchronization with the accessors, and destroy the Future object. This number was extrapolated from the data presented in Figure 3. We multiplied the workload by the number of Futures used in each run, and then subtracted that from the average wall-time of the trial. We divided that number by the number of Futures invoked in the trial to get the overhead per Future for each set of parameters:

$$overhead = \frac{avg.\ wall\text{-}time - (workload * futures\ invoked)}{futures\ invoked}.$$

As can be seen in Figure 3, the performance curves quickly reach a saturation point when the number of competing OS-threads becomes significant. This is primarily due to contention on the thread queue scheduler. As the number of OS-threads grows, the contention on the thread queue scheduler also increases, due to a higher number of concurrent searches for available work. This increased contention occurs in both global queue schedulers (where all OS-threads poll the same work queue and must obtain exclusive access to said queue

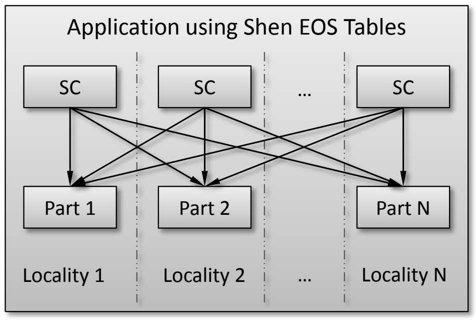

Figure 4: Schematic of an application using the distributed partitioned Shen equation of state (EOS) tables. Each locality has a Shen EOS client side object (SC) allowing to access all of the table data transparently. At the same time, the Shen EOS table data is partitioned into chunks of approximately equal size, each of which is loaded into the main memory of one of the localities (Part 1 ... Part N), thus lessening the required memory footprint for each of the localities.

for some period of time) and to some extent in local queue schedulers (where work stealing occurs, which causes queue contention, albeit to a lesser degree than in the global queue scheduler). As we increase the workload in each Future, OS-threads spend more time executing the workloads and less time searching for more work. This decreases contention on the queues. Adding a new OS-thread is beneficial as long as the contention overhead that it causes is not greater than the parallel speedup that it provides.

5.3 The Overhead of the Shen EOS Table Partitioning

We created an HPX component encapsulating the minimally overlapping partitioning (ghost zone of single element width) and distribution of the Shen EOS tables to all available localities, thus reducing the required memory footprint per locality [19]. A special client side object ensures the transparent dispatching of interpolation requests to the appropriate partition corresponding to the locality holding the required part of the tables (see Figure 4). The client side object exposes a simple API for easy programmability.

The second part of this section describes the setup and results of the measurements we performed in order to estimate the overheads introduced by distributing the Shen EOS tables across all localities. To evaluate the scalability and associated overheads of the distributed implementation of the Shen EOS tables, a number of tests have been performed, all of them with a fixed number of total data accesses (measuring strong scaling). The tests have been run on a different number of localities and with varying numbers of OS-threads per locality. The current HPX implementation supports only a centralized AGAS server that may be invoked in two configurations: either as a standalone task on a dedicated locality or as a part of one of the user application tasks. Our tests used a standalone AGAS server, firstly to avoid interfering with the user workload and secondly to eliminate the generation of asymmetric AGAS traffic on localities hosting data tables. Unlike the client applications, the AGAS server used a fixed number of OS-threads throughout the testing to ensure that sufficient processing

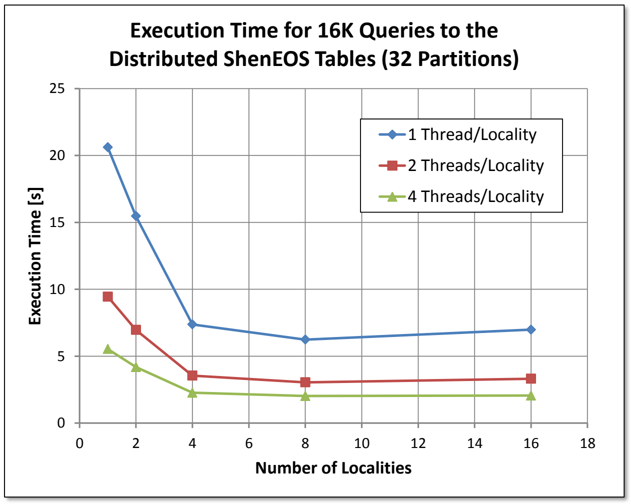

Figure 5: Scaling of the execution time for 16K separate non-bulk data interpolation queries to the Shen EOS tables distributed across 32 partitions measured for different number of localities and varying number of OS-threads per locality.

resources are available to the incoming resolution requests.

The tests were performed on a small heterogeneous cluster. The cluster consists of 18 localities (excluding the head node) connected by Gigabit Ethernet network. Two of the machines are 8-socket HP ProLiant DL785s, with 6-core AMD Opteron 8431s and 96 GB of RAM (533 MHz DDR2). The other 16 localities are single-socket HP ProLiant DL120s, with Intel Xeon X3430s and 4 GB of RAM (1332 MHz DDR3). All machines run x86-64 Debian Linux. Torque PBS was used to run multi-locality tests.

Figure 5 shows the execution times collected for the data access phase with a special test application executed on up to 16 localities and 1, 2, and 4 OS-threads per locality. The total number of distributed partitions was fixed at 32 to preserve the AGAS traffic pattern when run on a different number of localities; all partitions were uniformly distributed across the test localities. The number of separate, non-bulk queries to the distributed Shen EOS partitions was set to a fixed number of 16K. Each of these queries created a Future encapsulating the whole operation of sending the request to the remote partition, schedule and execute a HPX-thread, perform the interpolation based on the supplied arguments for the Shen EOS data, sending back the resulting values to the requesting HPX-thread, and resuming the HPX-thread that was suspended by the Future in order to wait for the results to come back.

The graph demonstrates that the overhead of distributed table implementation does not increase significantly over the entire range of available localities. While the scaling is much better when the number of localities remains small (up to 4), the overall time required to service the full 16K data lookup requests remains roughly constant. The test application itself does not execute any work besides querying and interpolating the distributed tables, which does not leave much room to overlap the significant network traffic generated with useful computation. This causes the scaling to flatten out beyond 8 localities. Using the distributed tables

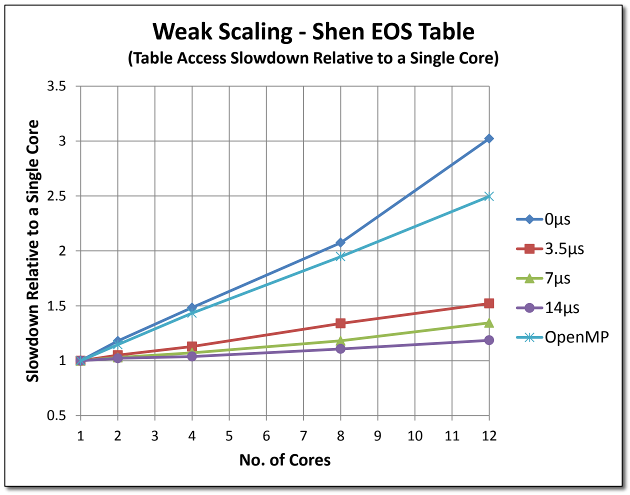

Figure 6: The relative slowdown in table access when run across various numbers of cores on a shared memory machine. For comparison, results using OpenMP are also provided; all other results use HPX for table access. When no additional work is overlapped with the table access, the slowdown relative to that seen on a single core can reach as high as a factor of 3; however, when other workload is overlapped with the table access, the contention in table access is increasingly amortized. These results used the smaller (288 MB) table.

in real applications that do much more work will allow for further amortization of the introduced network overheads. The results also imply that a single AGAS server is quite capable of servicing at least 16 client localities, especially considering the intensity of request traffic over the Ethernet interconnect deployed in out testbed. We plan to further evaluate this aspect of distributed table implementation using faster interconnect networks, such as Infiniband.

6. RESULTS

Accessing a single, potentially distributed Shen equation of state table using multiple threads for converting conservative variables to primitives and vice versa as required for the flux-conservative HRSC method results in a slowdown when compared with using multiple independent copies of the table. There is also some additional overhead in using Futures in the tabular access. In Figure 6 the table access slowdown relative to a single core on a shared memory machine is presented. The results are a weak scaling test where the results for each workload have been normalized to the corresponding single core performance. In this test, each core accesses and interpolates 64k unique values in the table as a single bulk operation. Using HPX on a single core of an Intel Xeon X5660 processor, this test takes 0.0728 seconds; for comparison, using the Fortran codes provided at [2] access and interpolation of the exact same 64k values takes 0.0549 seconds, reflecting the increased overhead in using HPX. As the number of cores accessing the same table increases, the access performance degrades. The primary reason for that is the competition of hardware threads executing on the same processor for access to memory, since most of the interpolation requests cannot be satisfied solely from processor caches

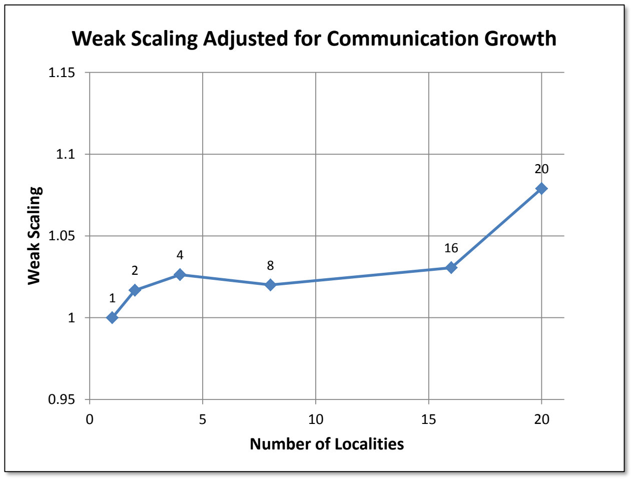

Figure 7: Weak scaling results using the 5.9 GB table based on the latest Shen equation of state [30]. The numbers above the data points indicate across how many localities the table was distributed. Each locality consists of two quad-core Intel Nehalem (2.8 GHz) processors. The interconnect between localities was Gigabit Ethernet. The workload provided was only that of table access and interpolation of 64000 values per locality. The results have been scaled to account for the linear growth in communication as the number of localities increases for this problem so that the computation and communication workload remain constant on each locality for each point in the plot.

(for the machine used in test, the ratio of utilized fraction of EOS dataset to the aggregate size of L3 caches was about 5). This is compounded even further by the fact that accesses are sparse and random in nature, and therefore result in decrease of the effective memory bandwidth. When no additional work is overlapped with the table access, the table access slowdown relative to a single core can reach as high as a factor of 3, an unacceptably high number for neutron star simulations. However, by overlapping work with the table access futures, the slowdown relative to a single core becomes much more reasonable.

For relatively small tables like the 288 MB table explored here, the memory cost of reading in a table for each core might be a manageable strategy in order to avoid any higher overhead costs associated with sharing the table using futures. For very large tables, however, there is no other viable option: the table would have to be distributed across several nodes and shared using futures. Using the recently released improved Shen equation of state [30], we have created Shen tables 5.9 GB in size for neutron star simulations. The increased table resolution contributes to improving the accuracy and robustness of neutron star evolutions as well as includes the most recently improved RMF results.

In Figure 7 distributed weak scaling results for the 5.9 GB table are presented. In these results, each locality consists of two quad-core Intel Nehalem (2.8 GHz) processors connected with Gigabit Ethernet. The table was distributed across as many as 20 localities and each locality interpolates 64K different table queries. The growth in communication due to distributed table access grows linearly as the num-

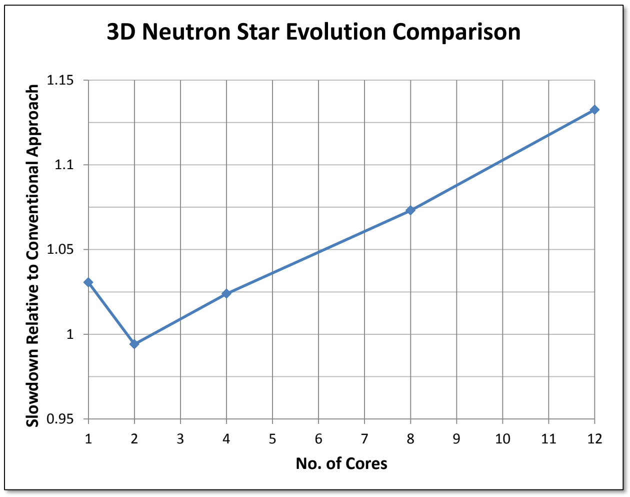

Figure 8: The relative slowdown in running a 3-D Neutron star evolution with a finite temperature equation of state for 10 iterations on a shared memory machine comparing Futures based table access with the traditional approach - reading in the table for each core. The table used in the comparison is the smaller (288 MB) Shen table. The simulation used a uniform grid with 50^3 points across the computational domain. We find the relative slowdown in using Futures based table access compared to reading in the table on multiple cores to be extremely minimal – at worse a factor of 1.13 when table access is shared across 12 cores – with a considerable savings in memory. These results resemble the 14 μs workload line seen in Figure 6.

ber of localities increases. For example, when the table is distributed across two localities, each locality will need to generate parcels to get $N/2$ nonlocal data points where N is 64K; when distributed across three localities, each locality will need to generate parcels to access $2N/3$ nonlocal data points; for four localities, that number becomes $3N/4$ for each locality, and so on. Hence while the workload on each locality is 64K interpolations, the overall communication grows linearly with the number of localities across which the table is distributed. The results in Figure 7 have been scaled to account for the growth in network communication so that the computation and communication workload remain constant on each locality for each point in the plot. This was done as follows: the baseline performance was measured on a single locality; performance and the exact number of parcels generated on the non-AGAS localities were measured for each case involving more than a single locality; weak scaling for those cases with more than one locality was then determined by

$$S = 1 + \frac{M - B}{BP}$$

where S is the weak scaling reported, M is the multiple locality performance time, B is the baseline performance time on a single locality, and P is the number of messages for a non-AGAS locality. We note that P is between 48 and 121 for the results on multiple localities presented in Figure 7 since messages are bulked together for improved performance; message sizes varied as the number of localities increase. All other weak scaling results presented in this paper compute weak scaling in the traditional way

$$S = 1 + \frac{M - B}{B} = \frac{M}{B}$$

since they were not distributed.

In Figure 8 we evolve a neutron star with the Shen equation of state on a shared memory machine (Intel Xeon X5660, 12 cores) for 10 iterations using a grid with 50^3 points across the computational domain and compare performance between the Futures based table access approach and the conventional approach of reading in a separate table for each core. By necessity this comparison had to use the smaller (288 MB) Shen table because using the larger table would have exceeded the memory available when testing the conventional approach. The Futures based table access performs extremely well compared to the conventional approach with negligible slowdown on up to 4 cores. At worst, when the table is shared across 12 cores, the slowdown is a factor of 1.13, consistent with the numbers presented in Figure 6 for the 14μs workload case.

7. CONCLUSION

We have examined finite temperature tabulated equation of state access in the context of neutron star simulations using a relatively new execution model called ParalleX. Using Futures to manage asynchrony, amortize contention, and hide network latency, we have presented a strategy for performing neutron star evolutions using extremely large tabulated equations of state with minimal performance and memory cost. Using the Futures based partitioned Shen EOS table access the slowdown compared to the conventional way of accessing these tables is less than $\sim$15%, and often much less than that. This added cost is justifiable, since as larger tables become available in simulation efforts, astrophysics simulations can then achieve a more realistic description of hot nuclear matter and incorporate more microphysics, including neutrino transport. Managing large tables in this asynchronous way would be difficult to implement when using conventional programming models, such as MPI.

Several key improvements to the results presented here are currently underway. As the HPX runtime system becomes NUMA aware, much of the memory contention observed here in both OpenMP and HPX runs can be eliminated [18]. Ways to reduce the Futures overhead reported in Fig. 3 even further are currently under investigation. All distributed runs presented here used Gigabit Ethernet interconnect; however, HPX support for the native Verbs interface for Infiniband is also underway. Hardware support for AGAS translation, whose first-cut implementation could utilize FPGA (Field-Programmable Gate Array) technology, promises to reduce key overheads, both in execution and storage, related to the software implementation. OpenCL support via percolation in HPX is also under development and could substantially impact the capability to perform neutrino transport in neutron star simulations.

8. REFERENCES

[1] LORENE web page. http://www.lorene.obspm.fr.
[2] STELLARCOLLAPSE web page. http://stellarcollapse.org/.
[3] 2012. http://www.etinternational.com/.

[4] M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. An application driven analysis of the ParalleX execution model. 2011.

[5] M. Anderson, E. Hirschmann, S. Liebling, and D. Neilsen. MHD with adaptive mesh refinement. *Class. Quant. Grav.*, 23:6503–6524, 2006.

[6] H. C. Baker and C. Hewitt. The incremental garbage collection of processes. In *SIGART Bull.*, pages 55–59, New York, NY, USA, August 1977. ACM.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[8] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. *CCS-TR-99-157*, May 1999.

[9] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[11] P. Colella and R. Woodward. *J. Comput. Phys.*, 54:174, 1984.

[12] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668, 1971.

[13] C. Dekate, M. Anderson, H. Kaiser, B. Adelstein-Lelbach, and T. Sterling. Improving the scalability of parallel n-body applications with an event driven constraint based execution model. *Int. J. High Perf. Comput. App.*, 2012.

[14] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP ’09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[15] D. P. Friedman and D. S. Wise. Cons should not evaluate its arguments. In *ICALP*, pages 257–284, 1976.

[16] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, 2007.

[17] P. D. Harten, A. Lax and B. van Leer. *SIAM Rev.*, 25:35, 1983.

[18] T. Heller, H. Kaiser, and K. Iglberger. Application of the parallex execution model to stencil-based problems. *International Supercomputing Conference 2012*, 2012.

[19] H. Kaiser. HPX Shen EOS SVN repository. Available under a BSD-style open source license. Contact gopx@cct.lsu.edu for repository access, 2011.

[20] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops*, pages 394–401, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[21] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA’93*, pages 91–108. ACM Press, September 1993.

[22] B. Lackey, K. Kyutoku, M. Shibata, P. Brady, and J. Friedman. Extracting equation of state parameters from black hole-neutron star mergers. i. nonspinning black holes. 2011.

[23] M. D. McCool, A. D. Robison, and J. Reinders. *Structured parallel programming patterns for efficient computation.* Elsevier/Morgan Kaufmann, Waltham, MA, 2012.

[24] P. Miller, A. Becker, and L. Kalé. Using shared arrays in message-driven parallel programs. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments, held in conjunction with IPDPS 2011*, Anchorage, AK, May 2011.

[25] C. Ott and E. O’Connor. A new open-source code for spherically symmetric stellar collapse to neutron stars and black holes. *Class. Quantum Grav.*, 27:114103, 2010.

[26] C. D. Ott, E. Schnetter, A. Burrows, E. Livne, E. O’Connor, and F. Löffler. Computational models of stellar collapse and core-collapse supernovae. *Journal of Physics: Conference Series*, 180(1):012022, 2009.

[27] J. Read, C. Markakis, M. Shibata, K. Uryu, J. Creighton, and J. Friedman. Measuring the neutron star equation of state with gravitational wave observations. *Phys. Rev. D*, 79:124033, 2009.

[28] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism.* O’Reilly Media, 1st ed. edition, July 2007.

[29] H. Shen, H. Toki, K. Oyamatsu, and K. Sumiyoshi. Relativistic equation of state of nuclear matter for supernova and neutron star. *Nuclear Physics A*, 637:435–450, July 1998.

[30] H. Shen, H. Toki, K. Oyamatsu, and K. Sumiyoshi. Relativistic equation of state for core-collapse supernova simulations. *ApJS*, 197:20, 2011.

[31] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS ’08, pages 277–288, New York, NY, USA, 2008. ACM.

[32] Stellar group. HPX GIT repository, 2012. Available under a BSD-style open source license.

[33] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. Preliminary design examination of the parallex system from a software and hardware perspective. *SIGMETRICS Perform. Eval. Rev.*, 38(4):81–87, Mar. 2011.

[34] K. Wheeler, R. Murphy, and D. Thain. Qthreads: An API for Programming with Millions of Lightweight Threads. In *International Parallel and Distributed Processing Symposium. IEEE Press*, 2008.