Parallel Applications for the Masses

Andrew Connolly, Jeffrey P. Gardner, and Cameron McBride

Abstract—With a new generation of astrophysical simulations and surveys on the horizon the astrophysics community is faced with the exciting challenge of how to process and analyze an apparently ever increasing data volume. Developing algorithms that scale with the size of data and utilize efficiently the massively parallel compute resources that are coming on line does, however, require substantial expertise in parallel algorithms. The steep learning curve associated with developing in this environment limits the number of astronomers who actually utilize Petabyte data sets there by hindering our ability to extract information from surveys in which we invest many hundreds of man years of work. We discuss here the development of algorithms for astronomy ranging from correlation functions to the tracking of moving sources that can be implemented on single and multiprocessor machines. We describe a framework, *N*tropy, that we have developed to ease the movement of serial algorithms to massively parallel systems. Our experience has shown that not only does our library save development time, it also delivers an increase in serial performance. Furthermore, *N*tropy makes it easy for an astronomer with little or no parallel programming experience to quickly scale their application to a distributed multiprocessor environment. By minimizing development time for efficient and scalable data analysis, we enable wide-scale knowledge discovery on massive datasets.

Index Terms—Parallel development tools, parallel libraries, massive astrophysical datasets, data analysis

1 Introduction

Astrophysics is witnessing a flood of data from new ground and space based telescopes and surveys. Virtual observatories (VOs) such as the National Virtual Observatory (http://www.us-vo.org) will federate all of these databases under all-encompassing umbrellas, making data access easier than ever before and providing the astronomer with a deluge of information. However, if we are to realize the full potential of these massive datasets, we must be able to explore, analyze and interact with them as easily, perhaps even more easily, than we could when they were small enough to fit on individual workstations. This is a daunting task, since the accession rate in data available to the astronomer is far surpassing the growth rate in speed of single CPUs.

On the other hand, multiprocessor platforms are becoming increasingly common. Computational clusters at department, university, and national levels are growing ever larger in size. Perhaps the most noticeable change to the average astronomer is the introduction of multi-core processor machines. Already, CPU manufacturers are offering 4 cores on a single chip. Intel has committed to building an 80 core chip within 5 years. Soon everyone's desktop or laptop machine will be a multiprocessor platform, some perhaps even massively parallel ones.

Concurrent with VO development and the progression towards increasingly wider parallelism is the construction of the protocols for interacting with distributed multiprocessor architectures. These tools will allow users with large numbers of small, independent tasks to quickly and easily distribute their workload to hundreds or thousands of processors. However, there will still remain a class of problems that cannot be trivially partitioned in this manner. These are cases where the entire dataset must be accessible to all computational elements and/or the elements must communicate with one another during the computation. Cluster finding, tracking, n-point correlation functions, new object classification, and density estimation are examples of problems that will require the astronomer to develop programs for multiprocessor machines in the near future.

- *Jeffrey P. Gardner is with the Pittsburgh Supercomputing Center, Pittsburgh, PA 15213 (e-mail: gardnerj@psc.edu)*
- *Andrew Connolly is with the University of Washington Seattle, WA and currently at Google, Pittsburgh PA (e-mail:ajc@phyast.pitt.edu)*
- *Cameron McBride is with the University of Pittsburgh Dept. of Physics & Astronomy, Pittsburgh, PA 15260 (e-mails: ajc@phyast.pitt.edu, cameron@phyast.pitt.edu)*

1.1 Shortening development time for parallel data analysis applications

Since their introduction in the late 1980s, massively parallel computers have demonstrated one thing: they can extremely time-consuming to program. After climbing the steep learning curve of parallel programming, the scientist can look forward to spending many times longer parallelizing their algorithm than it took to write it in serial. For this reason, the high-performance computation (HPC) community is largely dominated by simulations. Even if it takes 10 or 20 person-years to write a parallel simulation code, the economics still favor its development since it is typically reused for many years by many people. Data analysis, alas, does not work this way. Every scientist has their own analysis technique. In fact, it is largely what makes us each unique as researchers. For this reason, astrophysicists do not typically have the time or resources to develop analysis codes from scratch to run on national compute resources, or even smaller departmental clusters. For the full scientific potential of sky surveys to be realized, we need to create a way to facilitate the development process of data analysis codes on massively parallel distributed memory platforms (MPPs).

Procedurally, tree-based algorithms usually employ divide-and-conquer strategies that are relatively straightforward to parallelize. The difficulty for achieving high scalability emerges when the size of the dataset exceeds the memory capacity of a single computational node, and A) the tree walks span the domains of many nodes and/or B) nodes must update data up other nodes as the calculation progresses. In these scenarios, which are common in astrophysics, the time required to communicate between processors bogs the calculation down. Thus, we focus on enabling problems in this regime.

Our research has been to design an approach that exploits the fact that while the number of questions the astronomer may ask of the data is limitless, the number of data structures typically used in processing the data is actually quite small. In fact, most high-performance algorithms in astrophysics use trees and their fundamental data structure, and most specifically kd-trees. This is because they are typically concerned with analyzing relationships between point-like data in an *n*-dimensional parameter space. Therefore, our library, called *N*tropy, provides the application developer with a completely generalizable parallel kd-tree implementation. It allows applications to scale to thousands of processors, but does so in a way that the scientist can use it *without knowledge of parallel computing* thereby reducing development time by over an order of magnitude for our fiducial applications. Furthermore, *N*tropy is also highly efficient even in serial and provides a mechanism whereby the scientist can write their code once,

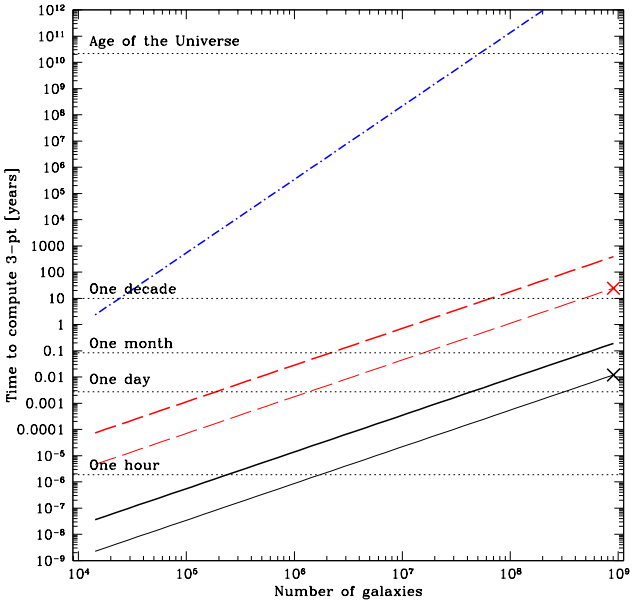

Fig. 1. Wallclock time required to perform a typical 3-point correlation function analysis on a dataset of galaxies vs. the number of galaxies in the dataset. The upper dot-dashed line is the naive algorithm that compares every particle combination and scales as $O(N^{2.8})$. The bold long-dashed line is the time required to compute using an efficient tree-based algorithm that scales, in the best case, as $O(N^{1.4})$ (with the worst case being $O(N^{1.8})$). The bold solid line shows the time required for the same calculation on 2048 processors. The non-bold lines indicate these same calculations done in the year 2012, assuming a doubling in CPU capability every 18 months. The stars show the estimated time required to process the LSST dataset of 1 billion galaxies in 2012.

then run it on any platform from a workstation to a departmental Beowulf cluster to an MPP. The scale of the computation is finally set by the scale of the scientific problem rather than the development time available to the researcher.

The last 2 decades have seen the development of many parallelization methodologies. Our goal in designing *N*tropy was to take the key components of a number of existent strategies and combine them into an intelligent implementation that enabled scientists to write highly scalable parallel tree-based applications in much less time than it would have taken them to write the same thing "from scratch." Therefore, the main focus of our research was not necessarily to create a novel parallel algorithm. What was new, however, was the experiment in combining several existing parallelization strategies under a single umbrella in a manner that was most useful for a specific target community. Our motivation for doing so was based on the success of the N-body cosmology code "PKDGRAV" [17], a highly scalable tree-based gravity calculator which has been in production for over 10 years and runs efficiently on a multitude of platforms, from small SMPs to MPPs with thousands of processors. PKDGRAV successfully combines several of the strategies that we will discuss below into a single application. The purpose of our research was to see if such a selective deployment approach could be extended from a specialized astrophysics application to a more general-purpose parallel tree library that was both highly scalable and straightforward for scientists to use.

In the following paragraphs, we discuss the pros and cons of several parallelization techniques. The literature offers a broad range of methodologies for efficiently parallelizing data trees, and the ones that we ultimately chose for *N*tropy are by no means unique. The general difficulty in assessing them is that most have only been tested on relatively small platforms (8 to 32 processors) whereas we are interested in scaling to thousands of computational elements. Thus, we largely restrict our discussion to strategies that are actively being deployed and benchmarked (with publications) on MPPs today. Our goal is to use the lessons they provide to design a methodology that leverages their strengths while avoiding their drawbacks. For the most part, codes on modern MPPs use message passing libraries, so our first metric evaluating *N*tropy will be to approach the high scalability of message passing. Our second metric will be to severely reduce the development time of a tree-based application in comparison to a purely message passing implementations.

1.2 Agenda-Based Parallelism

In an agenda-based approach, the programmer exists in a serial-like universe where he essentially writes serial code. Parallel speedup arises when certain tasks can be executed simultaneously by all processors. A common example is a DO loop. A loop of N iterations can be distributed over N_{PE} processors, with each processor calculating roughly N/N_{PE} iterations, provided that all iterations are independent from one another. The advantage is that the programmer need only write a serial pieces of code, and the compiler takes care of all of the gorey details of message passing and synchronization.

There have been many agenda-based compilers developed over the years, and it would be impossible to examine most of them however briefly. Nonetheless, it is possible to make some generic observations. One stumbling block for many was that they were designed for shared memory architectures, which are relatively rare these days. Another problem is that most have difficulty scaling beyond a few hundred processors, largely because they focus on turning a serial problem into a parallel one. The compiler proceeds along a single thread until it identifies instructions that can be conducted in parallel, whereupon it launches the parallel computation, then synchronizes at the end. The common obstacle to scalability in this approach is that parallel tasks end up becoming too "fine grained." Spawning tasks and synchronizing is expensive—sometimes they can require nearly 1 million CPU cycles on distributed memory machines (including NUMA systems)—and this paradigm demands that this be done frequently. Furthermore, between parallel regions there is often serial code, and Amdahl's Law tells us that any amount of serial code will rapidly squelch scalability. For these reasons, agenda-like codes are rarely seen running on the MPPs of today. Some examples of popular distributed memory compilers are Co-array Fortran, UPC, and HPF [11, 12, 3, 8].

Some agenda-based parallel compilers attempt to provide higher-level capabilities in order to increase granularity. A good example of this strategy is ZPL [2], which provides the programmer with a way to manipulate an n-dimensional shared array in a spatially aware manner. One can operate on the array as a whole: e.g. shift array elements along principle axes. One can also operate upon array elements conditional to their location in the array: e.g. add my value x to that of my neighbor above me if that neighbor has flag f =TRUE. By giving the programmer the ability to string together very high-level array manipulation commands, ZPL enlarges the granularity of the computation, allowing it to scale. Provided that your algorithm fits into this paradigm, ZPL is an excellent solution that may potentially scale to thousands of distributed processors. Unfortunately many scientific applications cannot be expressed using this formalism, and therefore find ZPL too restrictive. A common limitation of such high-level agenda-based approaches is that your problem must map onto the high-level instructions and structures that the compiler provides. In general, efforts thus far have worked well with regular arrays, but can be exceedingly cumbersome for algorithms that use irregular and/or adaptive data structures like trees. Nonetheless, efforts like ZPL demonstrate that it is possible to scale well using agenda-based parallelism provided that each high-level instruction in one's agenda maps easily onto the computation. We will revisit this observation when we discuss *N*tropy.

1.3 Explicit message passing

In contrast to agenda-based compilers, message-passing libraries provide almost limitless flexibility and generality. Because the programmer is in control of any interprocessor communication, he can use his insight into the algorithm to maximize its granularity and minimize the

effect of network latency. For these reasons, nearly all applications that run on modern MPPs use message-passing libraries. The most common library by far is MPI. Like most interfaces that offer a high level of control and generality, the drawback of MPI is that it forces the application developer to program at a very low level. This can be very time consuming if the thread domains are decomposed using structures more complex than regular grids, because it becomes difficult to use MPIs collective communication facilities. Furthermore, MPI poses an interesting paradox: even though MPI enables largely asynchronous execution, the more synchronous one's approach is, the easier it is to express it in MPI. Similarly, as one's algorithm approaches the ideal of few barriers and lots of asynchronous communication, it rapidly becomes quite challenging to implement in MPI. As we will demonstrate later, we designed *N*tropy so that it simplifies the process of writing an asynchronous application with minimal barriers.

1.4 Remote Method Invocation

One intriguing evolution of the explicit interprocessor messaging paradigm is ARMI, an advanced "remote method invocation" library for C++ [14]. RMI (or RPC for "remote procedure call") generically refers to a facility for launching procedures or methods on a remote processing element. Message passing libraries like MPI have a data-centric view of communication in that they simply transmit data from one location to another. RMI, on the other hand, means that you pass an executable procedure, usually accompanied by data, between physical locations. Note that each approach is essentially interchangeable: it is possible to package routines such that they can be passed via MPI (in fact, this is what ARMI does). Likewise, it is possible to use RMI to transfer data: if thread T needs to get data x from processor P's domain, for example, T would invoke a method on P that would return x. Some advantages of ARMI—which is written on top of MPI—is that it is conceptually cleaner than MPI, and it attempts to aggregate multiple remote invocations together into a single message, thereby reducing communication overhead. However, programming in ARMI still does not guarantee scalability. Navigation of adaptive data structures is typically a serial operation: one looks at a node or level of the structure, then uses that information to advance to another location, which must then be acquired. Therefore, message aggregation does not, in and of itself, help us in our quest for extreme scalability for tree codes. However, RMI does offer a straightforward mechanism for using an agenda-based approach to invoke *one's own functions* on multiple processors (rather than only those provided by the compiler).

1.5 Split-Phase Execution and Workload Virtualization

One compiler that has enjoyed some important successes is CHARM++, a parallel extension to C++ [5]. Like ARMI, CHARM++ also treats communication as the process of sending a methods, along with relevant data, amongst compute elements. CHARM++ differs from traditional RMI approaches in that it uses "split-phase execution": once a remote method is invoked, the invoking thread never receives a return value, nor can it check on the invokee's status. In order to accomplish a roundtrip message, for example, object A invokes object B. When object B completes its RMI, it must then reinvoke object A. In practice, however, the goal of split-phase execution is not to facilitate moving the data to where the computation is, but rather to make it easy to move the computation to where the data is. In other words, B simply carries on with the part of the calculation that needed the remote data and might not report back to A at all.

Split-phase execution complements CHARM++'s second important feature: process virtualization. In this paradigm, one typically creates 100 or 1000 times as many virtual compute threads as processors, and the threads migrate between processors redistributing workload as needed [6]. In our example above, when a object A invokes remote object B, it can elect to suspend itself until it is re-invoked by B. In the meantime, another object will execute. Therefore, a physical processor should always be busy doing productive work and never have to wait for messages to complete. This paradigm has proved successful in the implementation of NAMD[13], a molecular dynamics code that scales to thousands of processors.

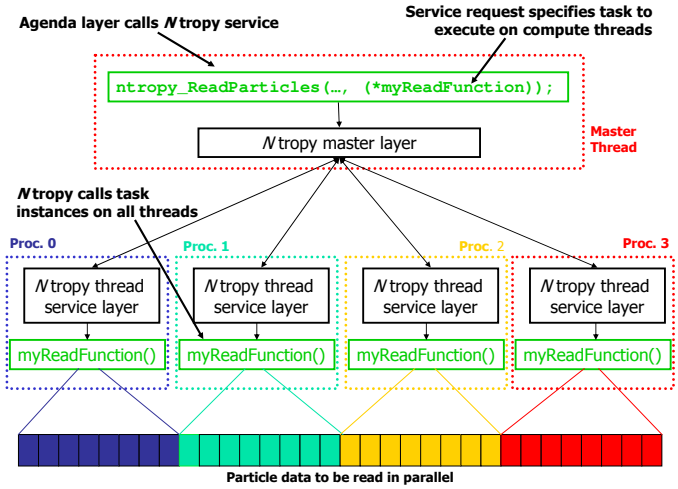

Fig. 2. How to use *N*tropy to read in a data file in parallel. The user writes the computation "agenda" which runs on the master thread only and steers the application. In the agenda, they call the *N*tropy service `ntropy_ReadParticles()` in which they specify a custom task `myReadFunction()`. Instances of that task are then launched on all threads.

Experience has shown that CHARM++'s split-phase execution strategy does not always mask communication overhead, however. The language is quite successful when each virtual thread (e.g. a "patch" of molecules in a molecular dynamics simulation) only needs to interact with a small number of other processors. In cases like this, the split-phase execution model of CHARM++ is an advantage to the programmer. Many scientific problems, however, demand that each computational task (e.g. a particle in an n-body simulation) access an large volume of data distributed across a very broad range of processors. Attempts to use CHARM++ for tree-based calculations, for example, became quickly saturated by network overhead because of the large number of messages that are spawned during the tree walk. In the end, reducing the number of messages turned out to be the deciding factor, not masking them with computation. The way to reduce the number of messages is to fetch needed off-processor data via a round-trip communication (as detailed above), then cache it locally for future requests by other virtual threads.

The problem with using the split-phase execution model for round-trip communication is that one must design each method in one's application so that it can be reinvoked at *every point* it requires an element of distributed data. For most algorithms, this demands substantial redesign. Therefore, for certain problems that must use round-trip data-centric messaging, split-phase execution can make the program much more difficult to write, not easier. Since *N*tropy is designed for tree walks, it focuses first on reducing network communication via data-centric messaging, then masking what remains. Since our goal is to make our library as simple as possible to use, we do not employ the split-phase RMI model. Process virtualization, on the other hand, is very useful concept to keep in mind for load balancing.

2 Methodology

Algorithms that distribute a tree across many computational nodes have historically proved to be among the most difficult to parallelize, because the most effective data structure for organizing the particles, a tree, is adaptive and irregular. Moreover, a typical treewalk often examines many tree cells that are spread across many processors. In order to achieve scalability, *N*tropy employs several data management techniques like caching of interprocessor data transfers, intelligent partitioning of the high-level tree nodes, and dynamic workload management. All of these capabilities are time-consuming to write from scratch. Using the *N*tropy library, however, the developer gets all of them for free. Consequently, we have been able to reduce the

time required to develop scalable parallel data analysis applications by over an order of magnitude. Furthermore, many of our *N*tropy applications actually perform *better* than competing efforts written with much greater effort from scratch. Our success proves that it is possible to build a general-purpose parallel library that is easy to use, efficient, and scalable.

2.1 Ntropy Structural Components

Fundamentally, *N*tropy is a library that provides communication and thread control infrastructure for parallel kd-tree computations. It incorporates a variety of concepts such as computational agendas, remote-method invocation (RMI), and message passing. The strength of *N*tropy is that it exploits each of these concepts when necessary and avoids them when they hinder scalability or usability.

The first piece on any *N*tropy application is the *agenda*, which serves as a computational steering mechanism and as an RMI launch pad for invoking parallel subroutines. An example of an *N*tropy service would be to read in data in parallel from an external file and is illustrated in Figure 2. In the agenda, which can be written in C, C++, or Python, the user calls `ntropy_ReadParticles(..., nParticles, fileName, (*myReadFunction));`. The *N*tropy infrastructure then invokes the method `myReadFunction()` on all of the compute threads. *N*tropy offers facilities for RMI on both generic and specialized routines. Our example uses the specialized interface `ntropy_ReadParticles()` that tells *N*tropy that the method the user is invoking is designed to read in `nParticles` elements of particle data from file `fileName`. This causes *N*tropy to do a little bit of extra work to make life more convenient. Each compute node calculates what its beginning and end particle will be, then allocates sufficient storage space. After that, each thread invokes the custom callback function `myReadFunction(fileName, startParticleID, nParticlesToRead, ptrToParticles)` which opens the file `fileName`, forwards to the particle `startParticleID`, reads in `nParticlesToRead`, and copies the data into the location pointed to by `ptrToParticles`. Once all instances of `myReadFunction()` have returned, the compute threads automatically signal completion to the master, which then returns from `ntropy_ReadParticles()`. *N*tropy's RMI facility makes the programmer's life much easier by furnishing a simple interface for coordinating parallel computation. The beauty of this approach is that it retains ease of workflow specification inherent in agenda-based compilers, but also permits customization at the per-thread level that maximizes the granularity of the computation.

2.2 Simplifying Access to Distributed Data

In principle, an RMI interface is general enough to also provide data transfer abilities. In practice, however, we have found that algorithms benefit greatly from a shared-memory view of distributed data. In other words, RMI is great for managing the flow of computation across nodes but, once those computations have been invoked, it is easier for the algorithm developer if they can be presented with an interface that makes distributed data behave as closely as possible to shared data. Furthermore, it is substantially easier to achieve high scalability if we treat methods and data differently, since we can reduce messaging activity through off-processor data caching (a capability that we discuss later). For these reasons, *N*tropy presents a separate, simplified mechanism for interacting with globally shared data on distributed memory machines: "simplified distributed data access" or SDDA.

In the above example, the instances of `myReadFunction()` do not have to communicate with one another. Now let us consider another task, `myTreeWalk()`, where a particle p_i needs to examine another particle p_j somewhere in the global dataset of all particles P. P is distributed across all processors, and p_i may or may not be physically located on the same processor as p_j. This is where the advantage of SDDA comes into play. In *N*tropy you can register certain blocks of memory to be "shared," i.e. visible to all threads. This causes *N*tropy to map all of the local blocks that were registered onto a single global address space that is accessible via a simple `Acquire()` command. In our example, if all threads register their local blocks of particles, then we can access any particle p_j in the set of all particles P simply `Acquire`-ing p_j. `Acquire()` will determine the physical location of particle p_j and conduct the necessary operations to retrieve that data wherever it resides, after which it simply returns a pointer to p_j.

The philosophy behind the SDDA approach is based upon the observation that the overwhelming majority of variables in scientific computations are local to each computational thread. Thus, each task is essentially a serial subroutine that only occasionally accesses globally shared data. Forcing the application developer to call functions to interact with shared data is therefore a minimal burden, but it does have the advantage of making it obvious to the programmer where potentially expensive interprocessor communication might occur. On the other hand, it is far easier for the developer to use our SDDA interface than to implement the data transfer by hand in a message passing library. Furthermore, we shall demonstrate that the *N*tropy SDDA layer offers many more performance advantages than a straight MPI call.

2.3 Achieving High Scalability

"Underneath the hood" of *N*tropy are two capabilities that substantially increase scalability: interprocessor data caching and dynamic workload management. These are features that are time-consuming for an application developer to implement themselves, but come "for free" when using our library.

2.3.1 Interprocessor data caching

When the application developer registers a block of shared data (as described above) *N*tropy logically maps that data onto cache lines. When an `Acquire()` call results in an off processor memory access, the entire cache line that holds the data of interest is fetched. The idea is that if the thread needed one piece of data, it will likely need the element next to it as well. Furthermore, future `Acquire()` calls for that same piece of data will not need to go off processor because the data will already reside in the cache. This mechanism results in fewer than *1 in 100,000* requests for off-processor data requiring a message to be sent in current *N*tropy applications.

At the moment, *N*tropy has two different kinds of caches: read-only and "reduction." The read-only cache is the simplest: the application is not allowed to write to shared memory blocks while the cache is active. The reduction cache is for data that is updated as the computation progresses, and is implemented in a non-blocking manner that requires no locks or other synchronizations, making it superior to other parallel concurrent-write mechanisms which must incur penalties to enforce cache coherency. The only constraint is that updates to the cache elements must be commutative and associative (similar to a parallel reduction operation). When a reduction cache is registered, the developer provides a reducer function, essentially the reduction operator, that takes as input the new value and the old value of the cached element, then returns a single new value. Nearly all read-write operations on shared data in scientific applications can function with within these constraints, and doing so alleviates all of the inefficiencies introduced by cache-coherency issues.

2.3.2 Dynamic workload management

Load balancing becomes increasingly crucial when scaling to thousands of processors. Most existing applications for massively parallel platforms use a predictive load balancing scheme where the application analyzes and distributes the entire workload before the computation progresses. This is a viable scheme for simulations—which comprise the overwhelming majority of MPP applications—since simulation volumes tend to have straightforward geometries, and the load-balancing behavior from the previous timestep can be used to extrapolate to the next one. *N*tropy applications, on the other hand, frequently have complex geometries (e.g. the Sloan Digital Sky Survey volume: SDSS, http://www.sdss.org) and, being data analysis operations, have no concept of a "time step." Consequently, a more advanced and dynamic load balancing scheme was required.

*N*tropy provides a facility to automatically migrate workload across processors as the calculation progresses, and is based on the process

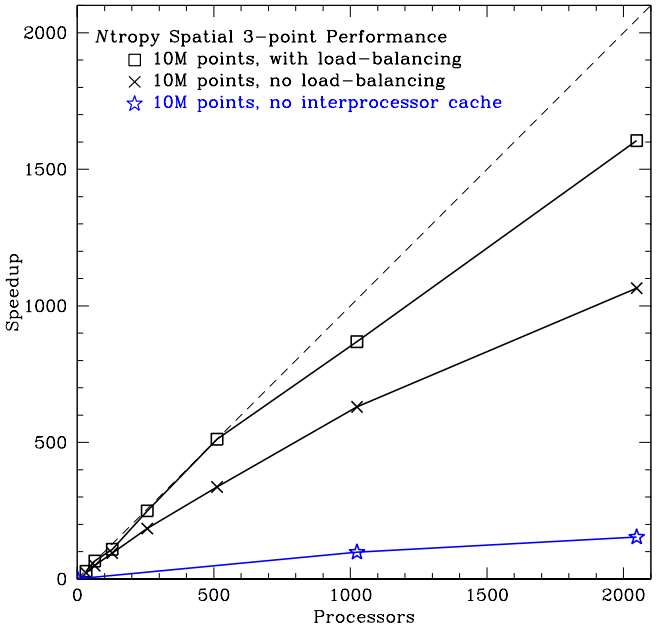

Fig. 3. The effects on scaling of interprocessor data caching and dynamic load balancing. The open squares show scaling for data caching and dynamic load balancing, while crosses demonstrate the effects of turning off load balancing. The open stars illustrate the further consequences of disabling the interprocessor data cache. This scaling test is for a single spatial 3-point calculation on a fixed problem size of 10 million particles randomly distributed in the Sloan Digital Sky Survey volume. It was performed on the PSC Cray XT3.

virtualization concept of CHARM++. Instead of invoking a single method instance per physical processor, the programmer can elect to invoke many more instances than processors. Instances of work are originally assigned to the processor that is predicted to have the most data relevant to that task. As the calculation progresses and a processor finds that it will soon run out of work to do, it requests more from a central workload manager. The thread with the largest remaining workload then donates several instances of its work to the requester. An important attribute of our work-management system is that a processor actually predicts when it will soon run out of work and initiates its request *before* this happens. The new work therefore arrives before the current workload is exhausted, and no time is wasted waiting for new work assignments. With a balanced workload, all threads are kept busy throughout the computation, and the overall time to solution is decreased. The advantage of our implementation to the programmer is that it takes place entirely "behind the scenes" within the library itself. Since the scientist does not have to recast their algorithm in a manner that supports split-phase execution, *N*tropy's workload managements system is very natural to use.

2.3.3 Performance diagnostics

Any performance-sensitive application should have diagnostic facilities for measuring performance and identifying bottlenecks. Although relatively straightforward in concept, details like timers and statistics gathering can be time-consuming to write. *N*tropy automatically records timing information for each task instance that executes, as well as for all I/O operations. Furthermore, the *N*tropy API makes custom timers available to the developer, who simply resets the timers and turns them on and off when appropriate. All timing measurements are then furnished upon request (to the desired level of detail) at the agenda level. *N*tropy also records detailed statistics on interprocessor communication and cache efficacy, making it easy to determine how much an application is being affected by communication latency.

3 Results

Two fully functional applications have been written in *N*tropy so far: an n-point correlation function calculator and a "friends-of-friends" (FOF herafter) group finder. Both applications difficult to parallelize, but for different reasons. The development time of each one was reduced by roughly a factor of 10 than if they had been written "from scratch" in MPI with similar performance: from 2 years to 3 months for n-point and from 8 months to 3 weeks for FOF.

*N*tropy was built using the RMI and data transport layers of the astrophysical n-body simulation code "PKDGRAV" [17]. We estimate the time required to develop from-scratch MPI n-point and FOF applications as roughly equal to the time needed to write the same parallel capabilities into PKDGRAV. The assumption is that for an MPI application to achieve the same level of performance as the *N*tropy n-point and FOF implementations, the necessary excess time would be roughly the same amount of time it took to write the MPI portions of PKDGRAV that are used by n-point and FOF. The development times for the *N*tropy implementations reflect how long would be needed for somebody reasonably proficient with the *N*tropy library.

3.1 N-point Correlation Functions

n-point identifies the number of n-tuples that can be constructed using particles in the dataset subject to spatial constraints. In 2-point, for example, one is interested in all pairs in a dataset that can be constructed from particles separated by a distance d, $d_{min} \leq d \leq d_{max}$. In 3-point, one seeks the number of triangles that can be made from points in the dataset where the sides (or angles) of the triangle satisfy certain configurations. There are two things that make this algorithm difficult to parallelize efficiently. Long-range spatial searches can examine lots of off-processor data, making it extremely latency sensitive. Furthermore, we are interested in the number of *unique* tuples, meaning that we must search the particles in a particular order, making the application difficult to load-balance. *N*tropy overcomes these obstacles by substantially reducing interprocessor messaging with its shared data cache and by automatically balancing the workload dynamically. Figure 3 shows the fantastic scaling that *N*tropy achieves. On thousands of processors, it scales 10 times better than the naive case.[1] The complex geometry of observational datasets such as SDSS prevents static load balancing strategies (which attempt to predict workload ahead of time) from scaling well. A 3-point calculation on the SDSS, for example, typically achieves about 50% ideal scaling on 2048 processors. Our dynamic load balancing scheme, on the other hand, automatically migrates work from busy processors to idle ones as the computation progresses and attains 80% scalability for the same calculation.

3.2 Astrophysical Group Finders

In a group finder like friends-of-friends, the difficulty is tracking groups that extend across processor domain boundaries. First, the groups of particles are constructed by using the kd-tree for spatial searchers. Then, the cross-processor groups are connected using an iterative graph-based procedure originally designed for shared-memory machines [15]. *N*tropy enables this algorithm by supporting user-defined shared irregular data structures like graphs, effectively mimicking a shared-memory architecture on a distributed machine. The shared-memory paradigm is, of course, much easier to program for, and it offers the scientist a broader choice of algorithms. For this reason, development of the group finder was substantially accelerated by using the *N*tropy library.

3.3 Serial performance

In addition to providing great parallel scalability, we found that the *N*tropy version of n-point actually ran 6 to 30 times faster than the existing widely-used serial implementation "npt" [7, 9]. This is because *N*tropy was written to be maximally efficient in serial as well. For

[1] Data points for the naive case are calculated from cache efficiency measurements of the cache-enabled runs which track total cache accesses, cache misses, and time penalty per cache miss.

example, *N*tropy arranges the tree nodes in memory such that a full-depth non-recursive tree walk (i.e. one that always descends the left child first until reaching a leaf node, then proceeds laterally) would access memory contiguously. This makes maximal use of cache and speeds up the tree walk. Furthermore, each tree node stores pointers to parent, children, and "next" nodes. A "next" node is the node to which a tree walk would proceed if it did not open either child. Thus, moving from one node to another requires following only a single pointer. Thus, by aggresively minimizing memory accesses, *N*tropy optimizes tree navigation and provides a high standard of serial performance.

4 Conclusion

We suspect that one reason most previous parallel development environments and tools have not achieved more widespread acceptance in the HPC community is that each one provided a single paradigm and forced every aspect of the application to conform to it. Our work with *N*tropy demonstrates that it is possible to take the effectual attributes of several parallelization approaches and combine them into a single facility that offers the developer a range strategies employable when appropriate. Specifically, our library provides the ease-of-use and scalability of agenda-based parallelism while providing as few constraints as possible on the algorithm by using RMI concepts to launch user-written subroutines on compute nodes. These subroutines are then provided with a shared-memory-like view of the computation which simplifies programming and enables many shared-memory algorithms. Instead of forcing the programmer to adapt their algorithm to a particular paradigm, *N*tropy offers several paradigms each adapted to the needs of the programmer, thereby providing an intuitive and natural solution to parallel application development.

*N*tropy's selective deployment approach and results also yield useful insights into the parallelization of data trees. The single largest problem faced by distributed tree implementations is communication overhead. A tree walk usually traverses a broad range of data and is largely unpredictable. If the tree is much larger than local memory, it is quite difficult to prefetch the data the walk is likely to need. Strategies like caching are therefore necessary and prove extremely effective at overcoming communication latency. A caching scheme can also efficiently update remote data as well, provided that the updates can be expressed in terms of a reduction. *N*tropy accumulates remote update directives locally until a cache line is flushed and sent to the remote node that owns the updated data. Process virtualization of divide-and-conquer schemes like tree walks can be highly effective for load balancing so long as they are implemented on top of a data caching mechanism. From an ease-of-programming standpoint, RMI is useful for providing an agenda-based approach to parallelism that still gives the programmer the necessary flexibility to implement their tree walk in a coarse-grained fashion.

*N*tropy facilitates the use of kd-trees on point-like datasets that are much larger than the memory of a single computational node. It enables the scientist to develop an application that scales to thousands processors in much less time that it would have taken them to write a similarly performing application with MPI. The tree implementation is also efficient and easy to use even for serial computations. *N*tropy therefore provides a seamless "upgrade path" for the researcher allowing them to run their application on any platform, from their workstation to a massively parallel supercomputer. By minimizing development time for efficient and scalable data analysis, *N*tropy enables wide-scale knowledge discovery on massive point-like datasets.

Acknowledgments

This work was funded by the NASA Advanced Information Systems Research Program grant NNG05GA60G and was facilitated through an allocation of advanced NSF–supported computing resources by the Pittsburgh Supercomputing Center, NCSA, and the TeraGrid.

References

[1] O. Babaoglu, L. Alvisi, A. Amoroso, R. Davoli, and L. A. Giachini. Paralex: an environment for parallel programming in distributed systems. In *6th ACM International Conference on Supercomputing*, pages 178–187, Washington, D.C., 1992.

[2] B. L. Chamberlain, S. E. Choi, S. J. Deitz, and L. Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[3] U. Consortium. Upc language specifi cations, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[4] http://www.sdss.org/.

[5] L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, New York, NY, USA, 1993. ACM Press.

[6] O. S. Lawlor and L. V. Kale. Supporting dynamic parallel object arrays. In *Java Grande*, pages 21–28, 2001.

[7] A. Moore, A. Connolly, C. Genovese, A. Gray, L. Grone, N. Kanidoris, R. Nichol, J. Schneider, A. Szalay, I. Szapudi, and L. Wasserman. Fast algorithms and effi cient statistics: N-point correlation functions. In *MPA/MPE/ESO Conference Mining the Sky*, 2000. xxx.lanl.gov/ps/astro-ph/0012333.

[8] A. Müller and R. Rühl. Extending high performance fortran for the support of unstructured computations. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 127–136, New York, NY, USA, 1995. ACM Press.

[9] R. C. Nichol et al. The effect of large-scale structure on the SDSS galaxy three-point correlation function. *Monthly Notices of the Royal Astronimical Society*, 368:1507–1514, June 2006.

[10] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprá. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, 2006.

[11] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[12] R. W. Numrich and J. Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, 2005.

[13] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kal, and K. Schulten. Scalable molecular dynamics with namd. *J Comput Chem*, 26(16):1781–1802, December 2005.

[14] S. Saunders and L. Rauchwerger. Armi: an adaptive, platform independent communication library. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 230–241, New York, NY, USA, 2003. ACM Press.

[15] Y. Shiloach and U. Vishkin. An o(log *n*) parallel connectivity algorithm. *J Algorithms*, 3:57–67, 1982.

[16] V. Singhal and D. Batory. P++: A language for large-scale reusable software components. In *6th Annual Workshop on Software Reuse*, Owego, New York, 1993. WISR.

[17] J. G. Stadel. *Cosmological N-body simulations and their analysis*. PhD thesis, AA(UNIVERSITY OF WASHINGTON), 2001.