Integrated Performance Views in Charm++: Projections Meets TAU

Scott Biersdorff, Allen D. Malony
Performance Research Laboratory
Department of Computer Science
University of Oregon, Eugene, OR, USA
email: {scottb, malony}@cs.uoregon.edu

Chee Wai Lee, Laxmikant V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA
email: {cheelee, kale}@cs.uiuc.edu

***Abstract*—The Charm++ parallel programming system provides a modular performance interface that can be used to extend its performance measurement and analysis capabilities. The interface exposes execution events of interest representing Charm++ scheduling operations, application methods/routines, and communication events for observation by alternative performance modules configured to implement different measurement features. The paper describes the Charm++'s performance interface and how the Charm++ Projections tool and the TAU Performance System can provide integrated trace-based and profile-based performance views. These two tools are complementary, providing the user with different performance perspectives on Charm++ applications based on performance data detail and temporal and spatial analysis. How the tools work in practice is demonstrated in a parallel performance analysis of NAMD, a scalable molecular dynamics code that applies many of Charm++'s unique features.**

***Keywords*-Charm++; TAU; Projections; NAMD;**

I. Introduction

High-level parallel languages seek to improve programmer productivity by providing rich abstractions for application development while hiding the low-level complexity of coding computations designed to run effectively on HPC platforms. However, there is a natural tension between more powerful programming environments, with their compilers, library frameworks, and multi-layered middleware, and the potential to achieve scalable performance on high-end systems. Generally, the further the developer is from the raw machine, the more susceptible the application code will be to performance inefficiencies. Furthermore, sources of performance behavior and possible performance problems become more difficult to observe and to map back to the high-level programming paradigm. It is therefore critical for performance measurement tools to be integrated in a seamless manner with the parallel programming system, and for knowledge about language abstractions, libraries, and runtime systems to be used to understand low-level performance dynamics.

There are several challenges to overcome. First, it is important to provide performance tools access to execution events of interest from different levels of language and runtime abstraction. These events can be used to trigger measurements that record performance metrics specific to the event semantics. The language runtime system should support event observation as part of its execution model. Second, to enable alternative techniques for different performance perspectives, it is necessary to build measurement interfaces and runtime support that can integrate multiple performance technologies. Third, performance analysis should attempt to map low-level performance data to high-level language constructs by incorporating knowledge of events and the computational model. The goal should be to identify performance factors to the application developer and language system at a level where the information can be most productively applied.

In this paper we present an integrated performance framework for the Charm++ [9][8] parallel programming system. Section §III introduces Charm++ and describes the design of its performance interface used to observe important performance events of interest. The performance interface allows Charm++ to be configured with different performance measurement and analysis capabilities. First the trace-oriented *Projections* tool is described then in §IV we discuss how the Charm++ performance interface can be leveraged to integrate a parallel profiling system. In particular, we look at configuration and use of the *TAU Performance System* [13] in Charm++ for profile-based performance measurement and analysis. We discuss issues of integration, runtime support, and overhead. Section §V demonstrates the advantages of using Projections and TAU on a Charm++ application for molecular dynamics, NAMD. Our experiments show how alternative measurement and analysis methods can be used to gain comprehensive performance insight for HPC applications developed using a high-level parallel programming paradigm.

II. Related Work

The challenge of integrating performance tools in parallel language systems has been faced in other projects. The Rice Fortran D language [14] offered a high-level data-parallel programming model that was portable to scalable clusters. Like High Performance Fortran, Fortran D allowed programmers to specify parallelism abstractly using data layout directives. The Fortran D compiler applied the directives to synthesize a SPMD program with explicit data placement, node-level parallelism, and message passing communication. The performance problem was how to interpret the low-level explicit measurements with respect to the source-level data-parallel abstractions. The Pablo performance environment was used to collect static and dynamic performance data provided through Fortran D compiler instrumentation [1]. With additional semantic information, Pablo could associate measurements with their high-level language generators in its analysis and presentation methods.

We faced similar challenges in the pC++ language system [3] where C++ extensions were used to specify collections of objects on which data-parallel methods could be applied. An early generation of the TAU system utilized information about collective identity to map performance data from parallel method invocation and message communications to individual collective instances. Because the pC++ method calls were synchronous, we could develop simulation tools to model scalability [12].

OpenMP's shared-memory parallelism directives and compiler-based translation posed problems for open performance tools because performance events were unexposed. The POMP research defined an event model and call-back interface for performance tools [10]. The OpenMP information provided by the API to configure measurement systems provides region context information to map thread-level measurements to parallel region locations. Both TAU and the Kojak/Scalasca performance tools have been successfully integrated in OpenMP programming environments using POMP.

III. Charm++ Performance Framework

Charm++ is a parallel object-oriented programming system based on C++. Charm++ is designed with the goal of enhancing programmer productivity by providing a high-level abstraction of a parallel program while delivering good performance on a wide variety of underlying hardware platforms. Programs written in Charm++ are decomposed into a set of parallel communicating objects. The Charm++ runtime system automatically maps sets of objects onto processors or threads across the parallel machine. Computation is triggered by the invocation of *entry methods* via messages on objects by other objects. This is similar to asynchronous Remote Method Invocation (RMI) except no return values are allowed. When a message for an entry method of an object arrives on a remote processor, the message is queued and eventually scheduled to start execution by that processor's Charm++ runtime scheduler. Once started, entry methods are executed to completion on a processor before another entry method on the same processor is allowed to start.

Figure 1 shows a logical view of interacting Charm++ objects and how these are subsequently mapped onto processors by runtime system.

A. Charm++ Performance Events

Gaining insight into a Charm++ application's parallel performance depends on what events are observable during execution. There are several points in the Charm++ runtime system's code base that can capture important events and information about their execution context. These include:

1) Start of an entry method.
2) End of an entry method.
3) Sending a message to another object.

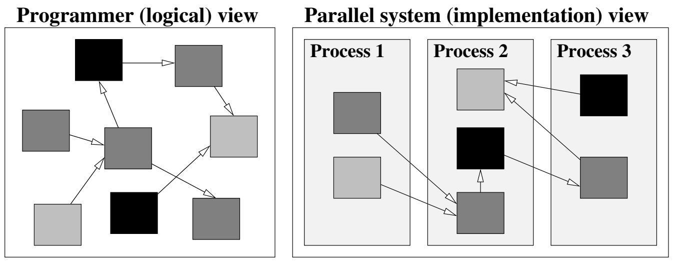

Programmer (logical) view
Parallel system (implementation) view
Process 1
Process 2
Process 3

Figure 1. Charm++ programming model.

4) Change in scheduler state: *active to idle* (entry method completes and no new message is available, *idle to active* (new message is available)

The first two relate directly to Charm++'s logical execution model. Observing message sends provides a runtime level perspective of object interaction. Scheduler state transitions expose resource-oriented aspects of the execution. The point is that the observation of multiple events at different abstraction levels is needed to get a full characterization of performance in a parallel language system such as Charm++.

B. Performance Call-back (Event) Interface

How a parallel language system operationalizes events is critical to building an effective performance framework. We use the term *performance event* to represent the execution of instrumentation code associated with any of the above points for purpose of making a performance measurement. The Charm++ performance framework implements performance events using a *call-back* mechanism, whereby instrumentation in the runtime system's code invokes any *performance module* (performance client) registered in the framework interested in the event. The framework forwards basic default event information, such as the event ID and time of occurrence, as well as other event-specific information to clients. As a performance module, a client will also have access to internal runtime routines and metadata not normally available to user code. These runtime routines and meta-data might then be used to derive more pertinent event-related information.

The framework exposes the set of key runtime events as a base C++ class (Figure 2). A new performance module like TAU (Figure 3) would inherit from this base class and implement methods for the interpretation and storage of information derived from individual runtime events. The client module need not handle every runtime event, merely the ones that are of interest to the module. The initialization of the performance framework initiates client module initialization through statically determined methods.

The call-back approach utilized by Charm++ has several advantages. Foremost, it separates concerns for performance event visibility from performance measurement. The call-back mechanism defines a performance event interface, but does not mandate how measurements are made. The ability to register different modules allows measurements to be configured for the desired performance experiment. It also allows the available measurement capabilities to be extended.

```
// Base class of all tracing strategies.
class Trace {
  // creation of message(s)
  virtual void creation(envelope *, int epIdx, int num=1) {}
  virtual void creationMulticast(envelope *, int epIdx,
               int num=1, int *pelist=NULL) {}
  virtual void creationDone(int num=1) {}
  virtual void beginExecute(envelope *) {}
  virtual void beginExecute(CmiObjId *tid) {}
  virtual void beginExecute(
    int event,    // event type defined in trace-common.h
    int msgType, // message type
    int ep,       // Charm++ entry point
    int srcPe,    // Which PE originated the call
    int ml,       // message size
    CmiObjId* idx) { } //index
  virtual void endExecute(void) {}
  virtual void beginIdle(double curWallTime) {}
  virtual void endIdle(double curWallTime) {}
  virtual void beginComputation(void) {}
  virtual void endComputation(void) {}
};
```

Figure 2. Simplified fragment of framework base class.

```
// TAU implements a performance module class inheriting
// from the base framework class.
class TraceTau : public Trace {
// Statically-determined method which is invoked by the
// framework at runtime initialization.
void _createTraceTau(char **argv)
{
  // TAU initializes the events buffer to hold 5000
        // different events ...
  bzero(events, sizeof(void *)*5000);
  // ...

  // TAU creates a new performance module instance and
        // attaches itself to the Charm++ tracing framework
  CkpvInitialize(TraceTau*, _trace);
  CkpvAccess(_trace) = new TraceTau(argv);
  CkpvAccess(_traces)->addTrace(CkpvAccess(_trace));
}

// Performance module constructor.
TraceTau::TraceTau(char **argv)
{
  if (CkpvAccess(traceOnPe) == 0) return;
  // TAU does more initialization, processes commandline
        // arguments ...
    // ...
    TAU_PROFILER_CREATE(main, "Main", "", TAU_DEFAULT);
    TAU_PROFILER_CREATE(idle, "Idle", "", TAU_DEFAULT);
    // ...
}

// TAU interprets Charm++ runtime events which are of
// interest to TAU.
void TraceTau::beginExecute(envelope *e)
{
  // ...
    startEntryEvent(e->getEpIdx());
  // ...
}
```

Figure 3. TAU integration with framework.

Registering multiple modules allows measurement techniques to be simultaneously applied.

It is interesting to note that a new measurement module can introduce additional requirements for event information, as seen with TAU. A call-back-based performance interface also allows Charm++ developers to update the call-back interface and/or runtime "performance support" library without affecting the other measurement modules.

C. Charm++ Performance Analysis with Projections

The *Projections* performance analysis tool is the original tool developed for Charm++. It utilizes the performance interface to collect performance measurements with two client module[1]:

- *summary* module: profile of message data volume
- *projections* module: full trace-based event logs

A detailed description of how the Projections tool is used to study the performance of, and tune an application can be found in here [7].

Projections parses the parallel event traces to generate multiple views for the analyst, who looks for performance problems such as load imbalance, unusually long entry methods, poor granularity, and/or communication bottlenecks. The *overview* display (Figure 5 Ⓐ) shows computational density (levels of work) versus each processor's data. This view can reveal possible load imbalance. The *time profile* view (Figure 5 Ⓒ) presents application performance in terms of how much work was performed for each entry method over time, summed over all processors. This view can be used to show if work from various phases of an application were successfully overlapped.

The *histogram* view shows the distribution of entry methods based on the amount of work performed within each entry method over a fixed time range. It helps us understand if the application exhibits poor work grain size distributions that could impact the effectiveness of load balancing or simply restrict the application's ability to scale. Lastly, the *time line* view (Figure 5 Ⓑ) faithfully reconstructs exactly what happened on each processor we wish to examine. This view is most useful for discovering communication bottlenecks and bad critical paths.

IV. TAU Integration in Charm++

While Projections provides powerful trace-based functionality for Charm++ performance analysis, the performance framework gives the opportunity to extend measurement support for profiling. The integration of the TAU Performance System®[4] is the first demonstration of measurement extension for the Charm++ programming environment. Below we discuss how this was accomplished and assess framework integration aspects, such as interface overhead and needed additions to the runtime system. However, it is also important to evaluate the utility of alternative performance tools, independently or together, and understand how best to apply

[1]Each module is activated by linking the user's application with the associated module library. The Projections module libraries are built by default and the TAU module is built when charm++ is given the location of the TAU distribution (released separately).

analysis techniques for the high-level programming paradigm. As parallel performance tools research shows, there are strong motivations overall for a range of techniques in performance problem solving. A simple early reason for adding TAU profiling support to Charm++ was to address a problem of trace buffer overflow for long running applications. Greater return on the investment in a flexible performance framework can be seen in the benefits of Projections and TAU integration for NAMD performance analysis (see Section §V).

A. TAU Parallel Profiling

TAU is an integrated parallel performance system providing support for instrumentation, measurement, analysis, and visualization for scalable parallel applications. The measurement system is cross-platform and provides both profiling and tracing support. Our interest for Charm++ is the integration of parallel profiling. The TAU's profiling model is based on the notion that every "thread of execution" in the parallel computation has an *event stack* that records the dynamic nesting of performance events marking the begin/end of interesting execution regions for measurement. Performance data is measured for events to reflect the *exclusive* performance (e.g., time) when the event was active (e.g., time spent in a method). The event stack allows *inclusive* performance data to also be kept (e.g., time spent in a method including time spent in nested method calls). For each event, TAU can collect performance data for execution time, hardware counters, and other metrics.

There are other features that come with the integration TAU profile measurements. For instance, *Phase profiling* enables profiles to be captured with respect to user-specified "phases of execution." Also, TAU's parallel profiling tools provide sophisticated analysis, database and data mining, and visualization capabilities. For more information on the TAU performance system, see [4].

B. TAU Performance Module

The problem of integrating TAU profiling can be seen as how to map the TAU profiling model onto the existing Charm++ runtime performance framework. TAU associates for each "thread of execution" an event stack (logically an *event tree*) with a top-level event (root) which encloses all other events. In normal TAU use, the top-level event can be thought of as the *main* routine of the program. In Charm++, each parallel process executes the scheduler as the top-level routine and the methods are called within this process. Given the performance events in the Charm++ performance framework, we might then see Figure 4 as a logical event transition diagram which could be profiled in TAU.

Following this approach, the TAU performance module was patterned on the Projections module to capture all method events. However, TAU requires a slightly different initialization to establish the scheduler creation as the top-level event. We distinguish the performance when the scheduler is active and processing messages ("Scheduler RUN" in Figure 4) as the *Main* event in the TAU profile. When no methods are executing and there are no messages to process ("Scheduler

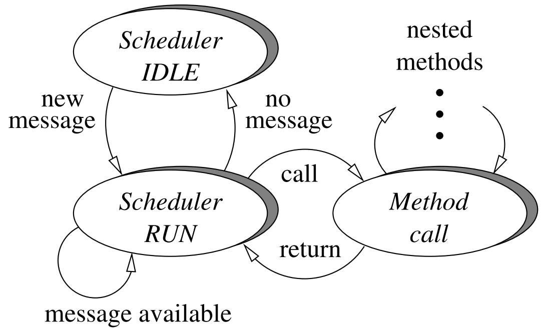


Figure 4. Charm++ runtime event transition.

IDLE"), performance data is associated with the *Idle* event. By the nature of how scheduler transitions are observed in the Charm++ runtime system for TAU profiling, there are slight differences in what *Main* performance covers relative to Projections. Method events appear naturally nested under *Main* in a TAU profile.

TAU can observe events outside of those generated by the Charm++ performance interface. These include user-level events created at the application-level, as well as, library-level events. In particular, TAU has the ability to measure MPI events through library interposition using a separate PMPI library included in TAU[4]. This allows the TAU performance module to track MPI communication[2] that occurs in the Charm++ system.

C. Performance Module Overhead

Before showing the TAU module in action with Projections on the NAMD application, it is important to test whether it is functioning correctly and assess its implementation efficiency. We created a simple benchmark program to measure module overhead under different instrumentation conditions. Both Charm++ and TAU have mechanisms to control the degree of enabled instrumentation (hence active events) and we wanted to also evaluate how overhead changed with greater measurement activity.

Table I shows the results from the overhead tests. We used different times bases and varied instrumentation level. Charm++ allows you to exclude entry events from the tracing system by use of the `[notrace]` entry method attribute. TAU's selective instrumentation works by runtime selection. In either case the performance frameworks callback routines are executed and they incur a small amount of overhead. See the NULL TRACE MODULE for the overhead when tracing framework is active but no events are actually being measured. We also show the overhead when both the projections and TAU modules are enabled, this is accomplished by having each performance module called individually, and thus results a greater amount of overhead. Projections and TAU have comparable overheads and these are relatively small. The availability of event selection options can also have beneficial effects in reducing overhead. Of course this depends on the specific Charm++ application being measured.

[2]available only when Charm++ uses MPI as its underlying layer for communication

Table I
OVERHEAD BY CHARM++ PERFORMANCE MODULE
(MICROSECONDS PER EVENT)

No measurement module	
Charm++ fully optizimized	0.09
Null trace module loaded	0.44
TAU module	
with [NOTRACE] option	0.55
with selective instrumentation	0.74
with fastest available timers	1.03
with GET_TIME_OF_DAY() timers	1.21
Projections module	
with [NOTRACE] option	0.49
with fastest available timers	1.99
TAU and Projections modules	
with fastest available timers	2.52

D. Profile Comparison

It is also important to compare the performance results from Projections and TAU. We can use the summary trace module to record some statistics on the entry event of a small molecular simulation included in Charm++. This will help us to understand what portions of the application the new events that were created by the TAU module are capturing. This information can also validate the correctness of measurements that the TAU module makes. Table II compares the summary and TAU modules. The TAU module records the inclusive percentage, exclusive time (in milliseconds), inclusive time (in milliseconds), number of time an events is called and how many events are call from each event as well as the inclusive time per call (in microseconds). The summary trace module records just the percentage of the runtime for each event and the number of calls made to each event.

As we can see the TAU module and summary module agree on the number of times the COMPUTE::INTERACT event was called. And the percentage of runtime accounted to the COMPUTE::INTERACT and the IDLE[3] events are about the same.

V. INTEGRATED ANALYSIS OF NAMD

The power of the Charm++ performance framework becomes apparent when the integration of Projections and TAU is brought to bear on the analysis of a complex Charm++ application. NAMD [5] is a parallel molecular dynamics code designed for high-performance simulation of large bimolecular systems. The computationally intensive part of NAMD involves computing interactions between atoms. Using Charm++, NAMD parallelizes these computations using a hybrid decomposition approach. First, NAMD spatially groups atoms into patch cells and distributes them across processors. This is known as the spatial decomposition technique. At the same time, NAMD employs force decomposition by creating "compute" objects to handle interactions between atoms of different patches. These "compute" objects need not reside on the same processors as patches. Bhatele et. al. [2] describes the NAMD parallel structure in detail.

[3]The Idle event's usage percentage was gathered from the Projections trace module.

Based on this approach, the path to good performance lies in the ability to distribute the computational workload across the parallel machine evenly while keeping communication overhead to a minimum. NAMD implements a load balancing framework to redistribute the "compute" objects across processors in order to maintain good performance as atoms can move from patch to patch over the course of a simulation run. A full-featured but expensive load balancing strategy is initially used in the simulation followed by strategies that refine any load imbalance after the simulation enters a steady state.

In reality, the performance achieved in a NAMD execution is dependent on several other factors, including the complexity of the bimolecular system, its size, the need for load balancing, and its cost. Some of these factors can be controlled by configuration parameters. Clearly, performance also depends on the parallel machine environment and its influence on the NAMD application. Providing a robust performance analysis of NAMD allows us to explore and understand the influence of these various factors on performance scaling.

We begin with performance experiments for a bimolecular model, *ApoA1* which is a simulation of a solvated lipid-protein complex in a periodic cell[11]. This is a relatively small model of 92K atoms which will demonstrate the performance impact of small computational grain at larger scale. Figure 5 shows three performance views from a Projections measurement of *ApoA1* on a 256-processor Cray XT3[4]. The Projections overview (view Ⓐ) shows computational density on each processor. The darker regions highlight periods of lower utilization. The Projections timeline (view Ⓑ) shows specific Charm++ activity in the zoomed time region, colored-code to reveal execution and performance behavior. (Note, only a small subset of processors are shown.) The Projections time profile (view Ⓒ) gives a statistical accounting of activity load across processors in time intervals. What these views convey are NAMD's performance dynamics and problematic features, such as periods of poor utilization.

Turning to a model ten times larger, we ran the NAMD *STMV* virus[6] benchmark with over 1 million atoms. To observe NAMD scaling performance, we choose to observe only a selected portion of the application, thereby removing from consideration the time spent starting up the simulation as well as writing out the results. We ran the simulation for 2000 steps during which the NAMD application turns performance tracing on using an interface call provided by the performance framework and then off again after an appropriate number of time steps.

For our scaling studies, we ran experiments on 256, 512, 1024, 2048, and 4096 processors on three different machines: BigBen, Ranger, and Intrepid[5]. We start with Projections to analyze the performance. Figure 6 shows time profile (top) and timeline (bottom) views for *STMV* runs on 1024, 2048, and 4096 processors on BigBen. We see for each execution two major periods of four timesteps in the NAMD algorithm. The first step in this time period involves *Particle Mesh Ewald* (PME) calculations where a series of FFTs are used to

[4]Pittsburgh Supercomputing Center's Cray XT3 (BigBen).
[5]Intrepid is an IBM BG/P at Argonne National Laboratory.

Table II
COMPARISON OF TAU AND PROJECTIONS SUMMARY

TAU						Name	Projections	
Time percent	Ex. msec	In. msec	Call	Subrs	In. usec/call		Usage percent	Call
100.0	0.005	1043802	1	1	1043802042	.TAU application	N/A	N/A
100.0	1947	1043802	1	272025	1043802037	Main	N/A	N/A
88.2	919591	920479	16856	12040	54608	Compute::interact	88.22	16856
7.7	80344	80344	36214	0	2219	MPI_Recv()	N/A	N/A
2.1	21434	21877	1456	100371	15025	Idle	2.008	N/A

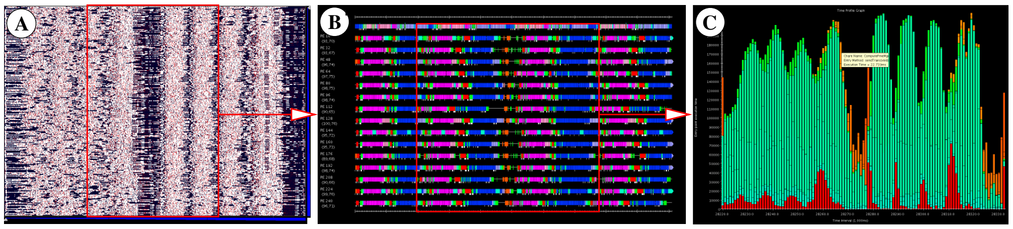


Figure 5. Projections *overview* Ⓐ, *timeline* Ⓑ, and *time profile* Ⓒ visualizations of a similar time region of a NAMD *ApoA1* simulation on 256 processors of PSC's Cray XT3.

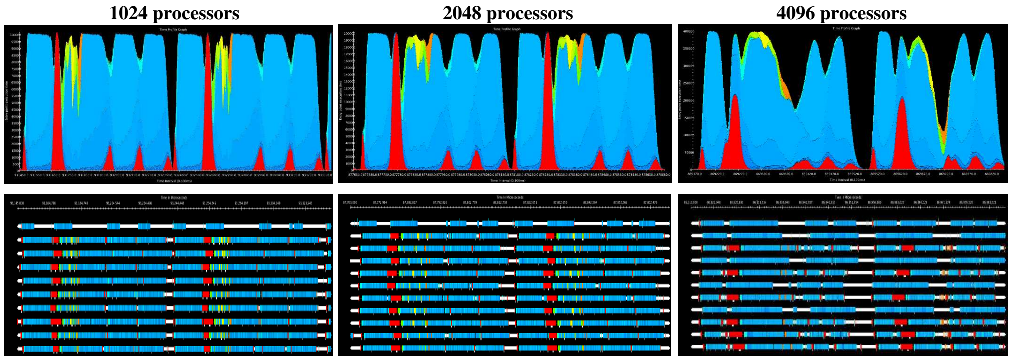


Figure 6. Projections visualization of a NAMD stmv simulation on PSC's Cray XT3 demonstrating the effects of scaling on NAMD's performance structure.

compute long-range electrostatic forces. PME computations in this experiment happens once every four timesteps and takes place in parallel on a subset of processors. The different colors indicate entry methods call during the PME time step: red - force integration, green/orange/yellow - FFTs along X/Y/Z planes with "pencil" parallelism. Three other times steps are apparent, but without PME calculations. There are two shades of blue, one for *enqueueWorkA* and *enqueueWorkB*. Notice how the FFT work is being overlapped[6].

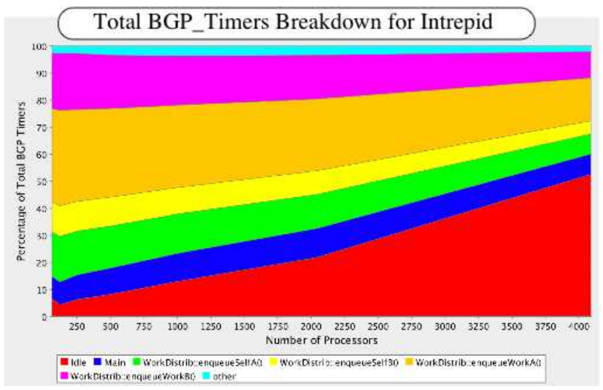
There are obvious changes in the shape of the displays as scaling occurs. The troughs between each time step becomes deeper and wider. Also, the PME work spreads out and takes longer relative to a time step. This causes other computational dependencies to be pushed out, resulting in a utilization depression prior to the second time step.

While Projections provides detailed views of a NAMD execution, we can use TAU profiles to see relative changes across executions. Figure 7 show a relative breakdown of mean performance for different NAMD events for Intrepid and Ranger from 64 to 4096 processors. It is interesting to see the relative performance effects with scaling and between the two systems, especially with respect to *Idle* (red) and *Main* events. Increasing *Idle* percentage signifies lower utilization, whereas as increasing *Main* percentage is capturing increased communication overheads. We can probe further in the profiles and compare scaling results for Ranger runs. Figure 8 shows that while computational work time is decreasing, scaling inefficiencies are being encountered.

Figure 9 compares the two systems on a 1024 NAMD execution. Oddly, the execution times for the computational work is opposite from what we expect. Intrepid's PowerPC 450 cores are roughly three times slower than Ranger's AMD Opterons (3.4 GFlops/core versus 9.2GFlops/core).

We initially suspected that excessive MPI communication overhead is showing up in the measurements of the computational events. An MPI measurement (not shown) confirmed this was not the case. However, by observing the "number of calls" metric recorded by the TAU profile (Figure 10), we noticed a ten-fold difference between the Intrepid data and Ranger data. This led us to check exactly how many NAMD time steps were actually being traced. It was discovered we were recording performance data for 1,000 steps on Ranger

[6]Although it appears as if there is less PME activity in the 4096 timeline display, it is an artifact of how the graphics are generated by Projections. Here only every 409th processor is shown. Some processors involved in the PME work are probably left out.



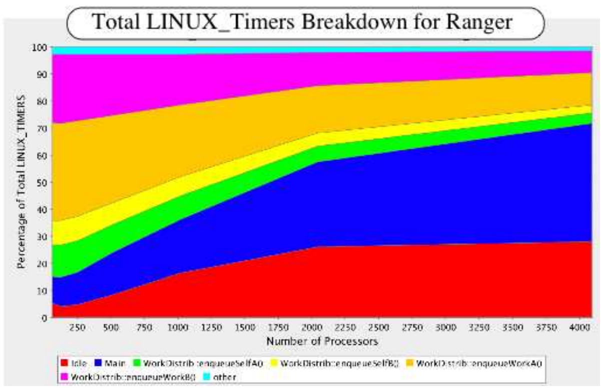


Figure 7. Relative Runtime Breakdown on Intrepid and Ranger showing the mean time across all processors which range from 126 to 4096. Time spent in Idle is represented in Red, time spent in Main is represented in Blue.

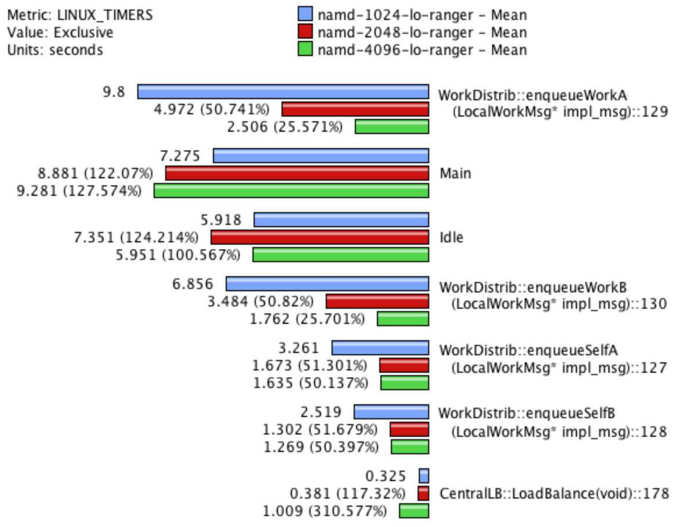


Figure 8. TAU profile showing scaling effects on the average total exclusive time per processor for the most time-consuming NAMD *stmv* events on Ranger.

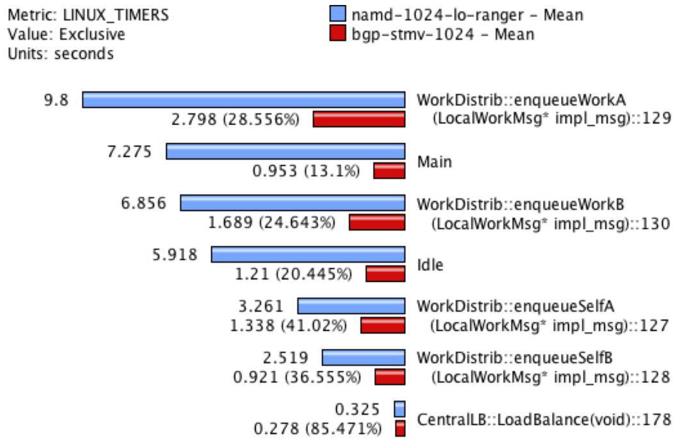


Figure 9. TAU profile showing the average total exclusive time per processor for the most time-consuming NAMD *stmv* events executed on Intrepid against Ranger.

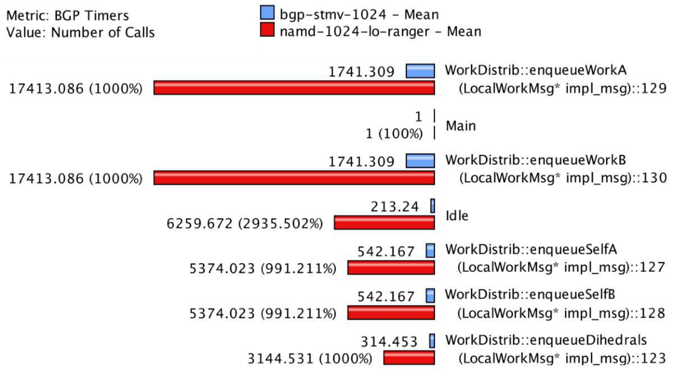


Figure 10. TAU profile showing the average number of calls per processor for the most time-consuming NAMD *stmv* events executed on Intrepid against Ranger.

and only 100 steps on Intrepid. With this information, the numbers now look reasonable – Ranger is taking 3 times longer to do 10 times more work. This performance reflects the fact that Ranger's processors are more powerful than Intrepid's.

Another piece of information revealed that the Ranger simulation became idle three times more often than Intrepid over the same number of steps. This appears to be in line with the expectation that the overlap of computation with communication will be better on Intrepid than on Ranger, again due to the differences in processor speeds. The basis for this expectation lie with the fact that Intrepid has slower processors than Ranger but possesses a slightly faster network interface.

We round off the study by quantifying the overhead the performance modules have on NAMD. We measured the instrumentation overhead on Ranger from 64 to 4096 processors. Figure 11 shows the overhead for NAMD using both the TAU module and the projections + summary module (in each case the projections trace buffer is set so that no overflow would occurred while NAMD was running.). We see that at large scale (4096 processors), where the granularity of work becomes small enough, the modules incur a nontrivial amount of overhead. While 10% overhead is within the range of most performance experimentation, further research could target overhead reduction in the context of this framework.

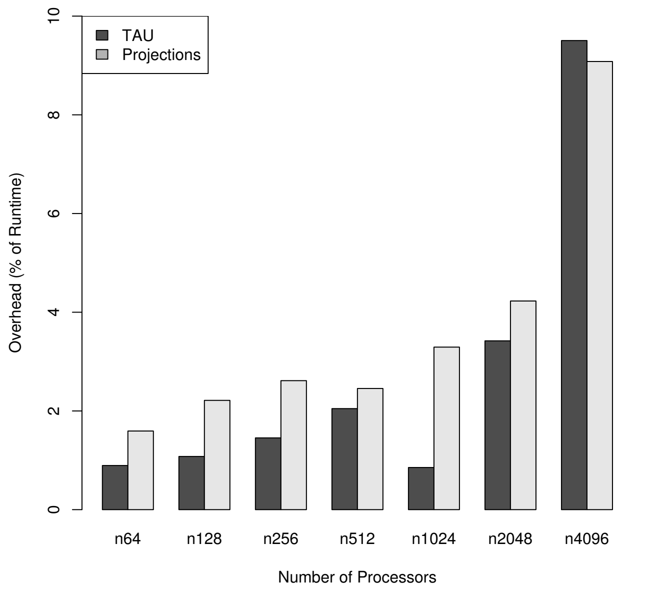


Figure 11. Instrumentation overhead for TAU and Projections.

VI. Conclusion and Future Work

The design of the high-level Charm++ language system includes a novel performance framework based on an open call-back interface which can convey Charm++ runtime events to performance measurement modules. We integrated the TAU performance system in Charm++ using the framework to demonstrate the power of the interface to extend performance experimentation capabilities. TAU complements the Projections trace-oriented and summary analysis with a parallel profiling perspective that exposes performance characteristics within and between application runs. We showed how Projections and TAU can be used together to investigate NAMD scaling performance on Linux and IBM platforms.

With the current implementation, we are in an excellent position to begin applying more of TAU's advanced profiling features in Charm++ performance analysis scenarios. For instance, TAU allows instrumentation of events at other levels not exposed by the call-back interface, including user-level code and MPI. These events will be captured in the profiles allowing better understanding of event relationships. Call-path profiling can be turned on in TAU to give further elaboration. Of particular interest is the integration of TAU's phase profiling. We should be able to use this effectively to separate performance profiles for different execution periods, both functional (e.g., load balancing vs. computation) and temporal (e.g., utilizing the time interval trigger for generating the Projections time profile.)

In some cases, more thought will be required to further integrate TAU and Charm++. Charm++ can execute in more sophisticated ways using multi-threading, process migration, and dynamic adaption. The performance framework will likely need refinement to support additional requirements placed on the performance modules. For instance, TAU can support shared-memory multi-threading already, but it is important to use the thread abstractions in Charm++ most appropriate for performance analysis. In general, the goal is to continue to support performance problem solving in association with the high-level programming system.

Acknowledgments

NAMD was developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology in collaboration with members of the Parallel Programming Laboratory, both at the University of Illinois at Urbana-Champaign. Access to the Ranger system at the University of Texas at Austin and BigBen at the Pittsburgh Supercomputing Center was granted through the NSF's TeraGrid. Access to Intrepid was granted by Argonne National Lab. The research was supported by the National Science Foundation, Software Development for Cyberinfrastructure (SDCI) (Grant No. NSF OCI-0722072). The authors thank Abhinav Bhatele, Philip Blood and Eric Bohm for sharing their expertise on running NAMD.

References

[1] V. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. Reed, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing '95: ACM/IEEE Conference on Supercomputing*, 1995.

[2] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[3] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel c++ runtime system for scalable parallel systems. In *Supercomputing '93: ACM/IEEE Conference on Supercomputing*, pages 588–597, November 1993.

[4] A. Malony et al. Evolution of a parallel performance system. In *Tools for High Performance Computing*, pages 169–190. Springer Berlin Heidelberg, 2008.

[5] J. Phillips et al. Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26(16):1780–1802, October 2005.

[6] Peter L. Freddolino, Anton S. Arkhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. Molecular dynamics simulations of the complete satellite tobacco mosaic virus. In *Structure*, volume 14, pages 437–449, 2006.

[7] L. Kale, G. Zheng, C.W. Lee, and S. Kumar. *Scaling Applications to Massively Parallel Machines Using Projections Performance Analysis Tool*, volume 22 of *3*, pages 347–358. Elsevier Science Publishers B. V., February 2006.

[8] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[9] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[10] B. Mohr, A. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for openmp. In *Los Alamos Computer Science Institute (LACSI) Symposium*, October 2001.

[11] James C. Phillips, Willy Wriggers, Zhigang Li, Ana Jonas, and Klaus Schulten. Predicting the structure of apolipoprotein a-i in reconstituted high density lipoprotein disks. In *Biophysical Journal*, volume 173, pages 2337–2346, 1997.

[12] K. Shanmugam and A. Malony. Performance Extrapolation of Parallel Programs. In *International Conference on Parallel Processing (ICPP '95)*, pages II:117–II:120, August 1995.

[13] S. Shende and A. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[14] C.-W. Tseng, S. Hiranandani, and K. Kennedy. Preliminary experiences with the fortran d compiler. In *Supercomputing '93: ACM/IEEE Conference on Supercomputing*, pages 338–350, 1993.