Performance Enhancements to Coven Through Multithreading

Nathan A. DeBardeleben
864-656-5909
ndebard@parl.clemson.edu

Vishal Patil
864-656-5909
vpatil@parl.clemson.edu

Walter B. Ligon III
864-656-1224
walt@parl.clemson.edu

Parallel Architecture Research Lab
Department of Electrical and Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915
http://www.parl.clemson.edu

Abstract

Coven is a framework for creating Problem Solving Environments (PSEs) for parallel computers. It has been used to create PSEs for satellite remote sensing, molecular dynamics, n-body simulations, and complex fluid dynamics. Coven provides the user with a component-based, pluggable environment to create, run, profile, and visualize output of parallel programs.

This work presents a collection of new features added to Coven that provide multithreading capabilities to applications without requiring additional programming by the application developer. These capabilities stem from overlapping computation, communication, and disk I/O.

A series of studies are conducted which look at potential performance gains this can provide as well as what are actually provided to a real application. It is shown that using these features can achieve an improvement of approximately 25% over a single threaded version of the same application. These gains require a very balanced workload, and for an unbalanced Fast Fourier Transform (FFT) application it is shown that performance increases as much as 18%.

Additional improvements to Coven such as garbage collection, performance profiling, distribution patterns, and several MPI-like collective operations are presented.

1. Introduction

Problem Solving Environments (PSEs) are becoming an integral part of modern high performance computing (HPC) due to the increasing complexity of the types of simulations being run and the underlying problems being modeled, as well as the increasing complexity of the computer systems employed. PSEs help to manage the complexity of modern scientific computing by hiding many of the details of the computer system, the application, or both behind a comfortable, familiar interface. A good PSE is flexible enough to allow the user to solve the problem yet powerful enough to provide reasonably high performance.

A number of PSEs have been developed including Khoros [10], BioPSE and Uintah[3] (based on SCIRun[8]), and CERSe[5] (based on Coven[4]). The Common Component

Architecture[2], SCIRun and Coven are problem solving environment toolkits that can be extended to build additional environments. In Khoros, modules (or *glyphs*) are connected to form dataflow graphs. Glyphs are separate, sequential programs which reads input from one or more files and write outputs to one or more files. In environments based on Coven, modules are subprograms dynamically linked which process data passed in memory.

While Coven is similar to SCIRun and CCA in many ways, the Coven project aims to study ways to add performance improvement to parallel applications developed with it. In previous work we introduced the structure of Coven and CERSe and showed how they are used to implement parallel applications. Overhead was also measured and found to be reasonably small[4]. In this paper we consider the use of Coven's multithreading capabilities to improve performance by implementing computation/communication overlap without requiring additional programming by the application developer. This mechanism is enabled by a number of features recently added to Coven including virtualization, performance profiling, and shared memory utilization.

In the next section we provide a brief overview of Coven and the relevant features. In subsequent sections we present our methods and experimental results showing speedup as a result of the application of multithreading. While it can be argued that the optimizations made are well known and can be implemented without the use of a PSE, most application writers would find them complex to program and Coven can provide the performance improvement without extra programming.

2. Coven

Coven is a framework for building problem solving environments (PSEs) for parallel computers. With Coven, users design modular parallel programs by writing modules in either C or Fortran and interconnecting them to form a dataflow graph. Modules contain special directives placed at the beginning of the code which describe the interface to the module.

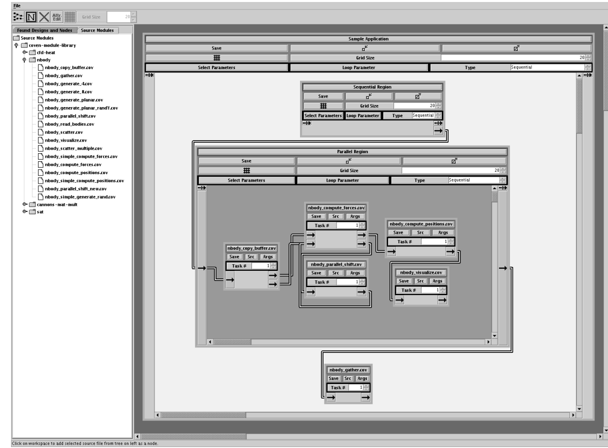

Figure 1. Graphical Editor Agent

Modules are combined to form applications using a custom Coven language that specifies how data flows through the modules. This language file is called a Coven program. A Java graphical front-end (Figure 1) can be used to graphically produce the Coven program. The Coven language can specify portions of the application that should run in parallel and specialized library modules effect inter-task communication. Applications are executed by a runtime engine that sequences the Coven language program, selecting modules to run and managing the flow of data. Figure 2 depicts the relationship between these components.

The runtime engine and GUI can both be extended for target PSEs. This extensibility makes it possible to customize Coven for a particular application domain. Coven has been used to create applications and PSEs in the following domains:

- Satellite remote sensing
- Molecular dynamics
- Complex fluid dynamics
- N-Body simulations

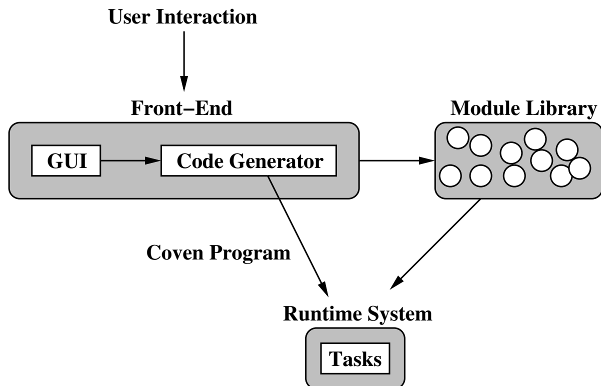

Figure 2. Coven Architecture

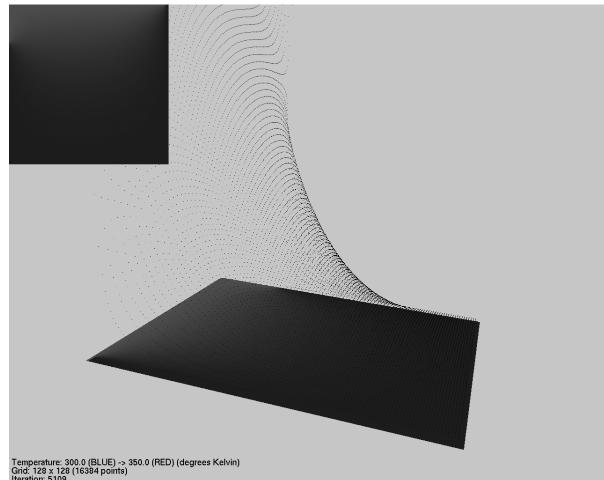

Figure 3. Open-GL 3-D Visualization Agent

Additionally, core algorithms such as the Fast Fourier Transform (FFT) and matrix multiplication have been implemented which are not application specific and can be used by any PSE.

Several Open-GL 3D visualizations have been built for Coven PSEs. Figure 3 shows a visualization for a fluid dynamics problem over a steel plate. Modules have been created which send data to this visualization at runtime or log data to a file for offline viewing.

2.1. Internal Implementation

All user data in Coven is encapsulated in an internal data structure called a Tagged Partition Handle (TPH). This structure is completely transparent to the user. TPHs are created by the engine, passed in and out of modules, and contain all data created by modules. This data is maintained in buffers which modules programmers interact with like arrays. Module designers do not have direct access to TPHs, instead they program by describing the flow of data throughout their application.

Many TPHs flow through the system concurrently. At any instant in time TPHs can be either processed by a module, sitting in a queue to be processed, or being sent to another task for processing. Before a TPH leaves a processor for another processor the TPH data structure is marshaled into a sequential stream of bytes which are transferred to the destination process using MPI communication operations.

Coven's runtime engine is composed of a collection of MPI tasks with a single task running on each processor. Figure 4 depicts a single processor with a program sequencer running on it. Modules run in each program sequencer and can communicate with other modules on other processors. Input and output queues exist before and after each process. TPHs are dequeued from the input queue, processed in the user-supplied modules, and collected back to a single point on a master process. Input queues are a recent addition to Coven and are described in Section 2.2. An optimization to Coven involving adding multiple program sequencers running on the same node is addressed in Section 4.

2.2. New Coven Features

Many improvements have been made to Coven since the version discussed in[4]. Some of the added features allow for new types of programs to work with Coven while others make programming Coven modules and writing Coven pro-

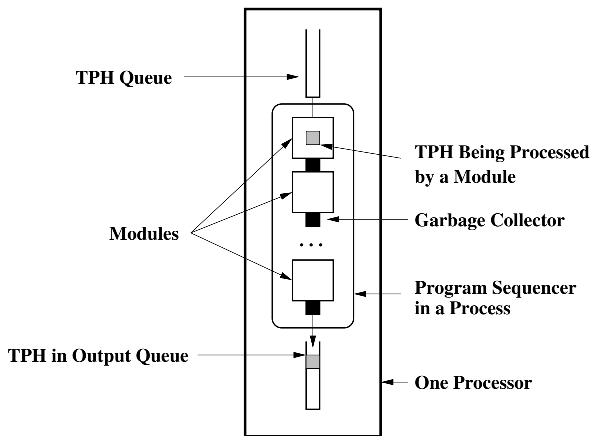

Figure 4. Coven Runtime Driver

grams simpler.

These features include:

Loops Coven programs can now be composed of loops of modules. The coven language supports `for` and `while` loops. Loop counters are available to modules inside of the loops.

Garbage Collection Data created by users in modules no longer needs to be specifically deallocated. Coven determines when a buffer will be last used by looking at the Coven program. After the last reference, Coven automatically frees the buffer.

Inter-Module Communication Coven now provides module programmers with a special MPI Communicator for use in communicating with another instance of the same module.

Scatter and Gather Users can now more easily specify the relationship between data and control where that data gets distributed through the use of scatter and gather modules. Coven provides datatypes similar to MPI datatypes for defining how data be scattered or gathered. Data partitioning schemes are available for scattering or gathering data in non-regular patterns. Coven provides block or cyclic schemes as well as a way for new schemes to be easily added.

Virtualized Partitions After data has been partitioned, multiple partitions may end up scheduled for the same processor. With virtualized partitions Coven allows those partitions to communicate. Coven provides send and receive operations which take as argument the partition to communicate with. Partitions are descheduled and kept in queues waiting pending communication operations. Once complete, they are allowed to continue execution on further modules. With this feature, Coven allows a module programmer to partition data into a distribution pattern without regard as to where the partition eventually gets placed. This creates a virtualized model of execution where a programmer may distribute partitions in a complex pattern but during inter-partition communication need not be concerned about where these partitions have been physically scheduled.

TPH Queues Coven now maintains queues of TPHs internally which it uses to keep TPHs waiting for processing as well as TPHs waiting to be received by another task. The maximum depth of these queues can be configured by the user which provides control over the balance between the amount of work each task can perform. If a task's modules take longer than another, controlling this queue depth can keep the shorter tasks from getting too far ahead and keeping the more complex modules from getting the processor. Other than configuring the maximum queue depths, all issues related to queues are completely transparent to the user.

These features allow many new types of applications to be implemented in Coven as well as provide some parameters that can be tuned to improve performance.

3. Performance Profiler and Debugger

Coven now includes a graphical, offline Java profiler to analyze Coven program execution. The profiler functions as both a debugging tool and a performance analysis agent. When a Coven parallel program is run, it generates a log file which describes the flow of the computation such as when did the modules run, where the TPHs were at different points in time, and information regarding memory usage, CPU load, and other measurements. This log file is then used by the profiler to generate visualizations for the various aspects of the program, thereby enabling the programmer to observe and analyze its behavior.

Coven uses the MPE (Multi-Processing Environment)[1] for logfile generation. In addition to logging MPI calls, using MPE it is possible to create user defined states and log these with additional data. Coven uses this facility to log module calls and auxiliary events. By using a standard logging format, other MPE logfile visualizers (such as Jumpshot[11]) can be used. The Coven profiler complements Jumpshot by presenting a view of the computation to the user in Coven-specific terms.

Currently the Coven profiler tool provides several different visualizations. Each of these visualizations is explained briefly below.

TPH Lifetime This visualization displays information regarding the flow of TPHs through various modules of the Coven program. This is basically a Gannt chart with the program time in seconds, on the x-axis and the threads on the y-axis. Module calls are represented as colored blocks with lines connecting them to indicate flow of a TPH through the system. With this view is is easy to see how a TPH flows between modules, threads, and processors.

Memory Usage This visualization displays information regarding the memory usage of different threads in the Coven program. This view can be helpful in identifying memory leaks and potential memory intensive modules.

Load Information This visualization displays information regarding the CPU load of each Coven module call. By looking at which modules have low CPU load, users can better determine which modules to place into which threads.

Collection of Bargraphs Four additional bargraph visualizations are provided which graph module runtime and auxiliary event runtime on both a per-processor and total-system basis. This can be helpful in easily identifying which modules are most time consuming and if any auxiliary events are particularly costly.

A user can customize each visualization at runtime to focus on specific events they are interested in. These include looking at specific processors or Coven threads, examining particular modules, or analyzing certain Coven internal events such as TPH queue depths.

4. Multithreading in Coven

Multithreading in Coven means running multiple threads of control on the same processor. We term these *Coven threads*. Each Coven thread on a processor runs a separate program sequencer and processes a separate set of modules.

Many new advancements have been added to Coven to facilitate multithreading parallel programs. Our test facility uses MPICH[7] as an MPI implementation. MPICH is currently not thread-safe, in that threads cannot use MPI calls to communicate with other threads. This feature is set to appear in a later version of MPICH. While some commercial implementations of MPI are thread-safe, widely used open source implementations such as MPICH and LAM-MPI are not.

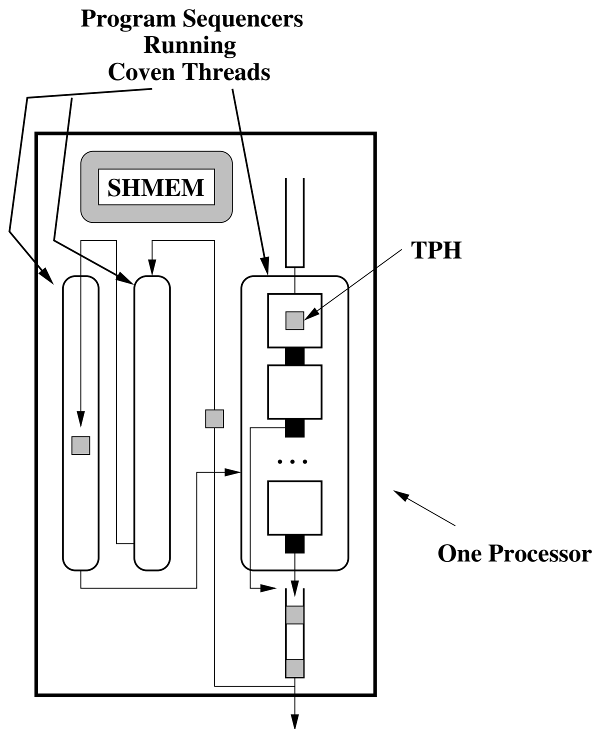

Figure 5. Multithreaded Runtime Engine

Figure 5 shows how the original runtime engine has changed to facilitate multiple threads. As a TPH flows from one thread to another the Coven runtime engine determines on which processor the thread runs and takes care of marshaling the TPH and transferring the TPH to that thread for processing. Once marshaled, a signal is sent to the destination thread signifying a TPH is ready to be transferred. The TPH is then placed in the output queue of the current thread and waits there until it has been accepted for delivery and successfully sent to its destination.

4.1. Program Language Augmentation

The Coven program writer may specify which thread a module runs in. A Coven program then often looks like Figure 6.

The `args` string is a C-style argument list of tags. The scatter and gather modules are executed in one thread of the master process while the parallel portion is executed in in two threads.

```
. . .
  thread_A fftw_scatter(args)
  PARALLEL {
    thread_B fftw_comp_nd(args)
    thread_C fftw_transpose(args)
    thread_B fftw_comp(args)
    thread_C fftw_transpose(args)
  }
  thread_A fftw_gather(args)
. . .
```

Figure 6. Thread Portion of Program

In this example the master process has only a single thread, Thread A. At runtime the use specifies the number of parallel processors to use, and on each one Threads B and C are spawned.

The `fftw_scatter` module partitions a 2D FFT into partitions for the number of selected parallel processes. Computation is performed in the `fftw_comp_nd` and `fftw_comp` modules. The `fftw_transpose` module is a network intensive component. Results are assembled on the master processor with the `fftw_gather` module. The goal here is to achieve an overlap between computation and communication by scheduling these types of modules to run concurrently.

To achieve this overlap it is clear that the initial decomposition must schedule multiple partitions of work on the same processor. In this way, when the computation portion is complete and the results are sent to the network communication thread, another partition can begin processing in the computation thread. For these types of programs, Coven provides the multithreaded capability in an attempt to achieve benefits from overlapping asynchronous operations.

4.2. Thread Implementation

Coven emulates multithreading by creating full-weight MPI tasks and scheduling them on the same processor. As mentioned in Section 2.2,

the Coven program sequencer maintains a user-configurable number of non-blocking receives which it uses to accept incoming TPHs to process. Therefore, at steady state each Coven parallel task is waiting in an `MPI_Waitany` for any of the receives (or possibly any non-blocking sends it has issued) to complete. MPICH 1.2 implements this method as a busy-wait, continually checking each request to see if it has completed. This operation causes the process to consume the CPU fully. This presents a performance problem when multiple MPI processes are running on the same processor, each at steady state sitting in an `MPI_Waitany`. MPICH2[9] (which is currently in beta testing) is the next generation version of MPICH and currently does not busy wait, thus MPICH2 was used for the results shown here.

4.3. Use of Shared Memory

Coven uses MPI to transfer TPHs between threads, both on the same processor and on remote processors. Testing showed that for large buffer sizes there could be a performance improvement by using shared memory for transfers between Coven threads on the same processor.

MPICH can use shared memory between MPI tasks on the same processor. When a task wants to send data to another task on the same processor, buffers are copied out of the source task's user space, into shared memory, and then back into the destination task's user space. Coven, on the other hand, uses the shared memory space for creating and managing all buffers that will be communicated with another Coven thread on the same processor. With this approach, Coven saves time by not performing costly copies of large buffers.

In an effort to demonstrate this feature a simple Coven program was created with two Coven threads running on a single processor. Figure 7 shows a screenshot of the TPH Lifetime view of the Coven Profiler described in Section 3. Each Coven thread runs a single module and the threads exchange a TPH back and forth twice. In this example, the TPH has a 256MB buffer in it. Notice that after the first module there is a space of 6.5 seconds where no module is executing. It is during this time that the TPH is being moved using MPI between thread 1 and thread 2. The total runtime for this program takes 33.9 seconds.

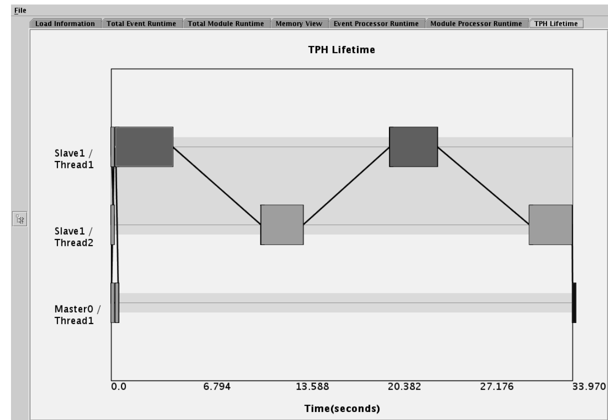

Figure 7. Without Shared Memory

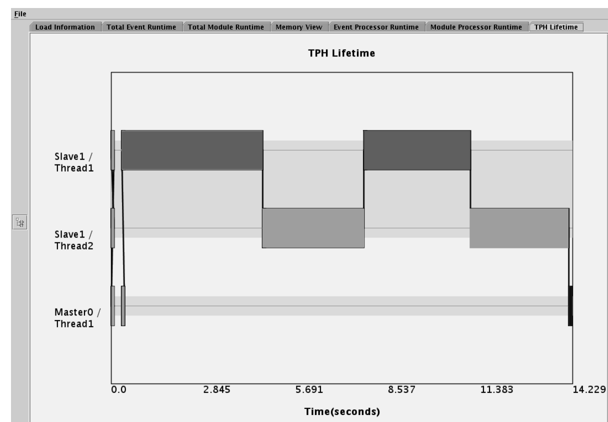

Figure 8. With Shared Memory

Figure 8 depicts the exact same Coven program with the Coven runtime engine using built-in shared memory. By using shared memory here, the second module begins nearly immediately after the completion of the first. The total runtime for this version is merely 14.2 seconds.

The user must specify the size of the shared memory region for Coven to allocate. Under Linux, this can be set with the `sysctl` command using the `kernel.shmmax` variable. Addition-

ally, the user can set the maximum size of a buffer may be in local memory before it is allocated into shared memory.

Whenever a user requests to create a buffer Coven determines if that buffer should go into shared memory or local memory. A semaphore is maintained between all Coven threads using System V IPC calls. All attempts to allocate or free data in the shared memory region are atomic across the processes. When a TPH is transferred from one process to the next, Coven determines if the destination process is local to the sending process. If it is, all buffers in shared memory are exchanged merely by passing the pointer to the shared memory address of the buffer. Memory management routines are provided for using the shared memory space.

All shared memory issues are completely transparent to the user. The two sample programs above differ merely by a flag in the Coven program source file stating whether to use or not use shared memory. While the user has the ability to tune the values of the shared memory buffer sizes and maximum local memory buffer size, a default is provided which rarely needs to be changed.

5. Case Studies

To analyze the performance effects of Coven's multithreading capability two sample applications were implemented in Coven. These applications were run on a 16 node Beowulf cluster consisting of nodes with one 1GHz Pentium III processor, 1GB of RAM, and connected by Fast Ethernet. Each node of the cluster runs Red Hat Linux 8.0, Linux kernel 2.4.20, and MPICH2 0.93. All programs were compiled with GCC version 3.2 using maximum optimizations.

5.1. Synthetic Application

A synthetic application was constructed in an attempt to look at several program variations and how they effect multithreaded performance. This application has a computation phase followed by a network communication phase. Each phase is implemented as a module with many tunable parameters which can be altered without effect on the other modules. While this is not a realistic application it allows the study of several interesting effects. The program was designed so that multiple units of work were assigned to the same processor. In this way, when a thread completed work on one portion of work and handed it to another thread it could begin processing the next unit of work.

Using Coven's profiler the CPU utilization of each module was analyzed. To determine the load each module placed on the CPU, Coven divides the total amount of time a module was scheduled on the CPU by the total amount of wall clock time that passed during the execution of the module. It was found that modules which load the CPU below about 60% were good candidates for placing into threads that run concurrently with other modules.

Additionally, the optimum performance was seen when a pipeline could be generated where the modules which ran in the pipeline did so for approximately equal amounts of time. The synthetic application was tuned in this way and the timings of the program runs can be seen in Table 1. The fourth column depicts the percentage improvement that using multiple threads was able to achieve. The gain ranges from about 24% to 28% for this application. The performance drops in the higher number of processors case due to the network communication phase increasing its load on the CPU. In particular, for this application the load for the communication module grew from 30% in the 2 processor case to 35% in the 16 processor case. Timings for 1 processor are not given since to do this study, interprocess communication was required.

Figure 9 shows a profiler screenshot of the single threaded version of this application running on 5 processors (1 master process and 4 slave processes). The computation module alternates with

Table 1. Synthetic Application Timings

# Procs	# Threads	Execution Time (secs)	% Improv.
2	1	231.81	-
2	2	167.42	27.7%
4	1	117.08	-
4	2	85.82	26.7%
8	1	59.15	-
8	2	44.06	25.5%
16	1	32.42	-
16	2	24.61	24.1%

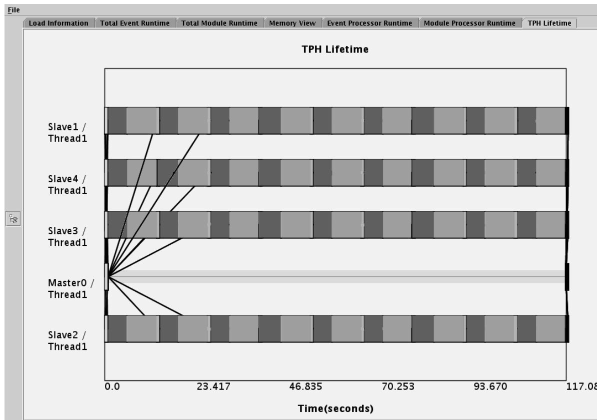

Figure 9. Synthetic Application Single-threaded

the communication module and it can be seen that the blocks take approximately the same amount of time.

This same application was placed into multithreaded mode by simply changing the Coven program language source file. A resulting profiler screenshot appears in Figure 10. In this case, the computation module ran in one thread while the communication module ran in another thread. Data passed between the two threads in a regular pattern. The total runtime of the application dropped from 117.08 seconds to 85.82 seconds, an improvement of 26.7%.

The balance between execution time and CPU

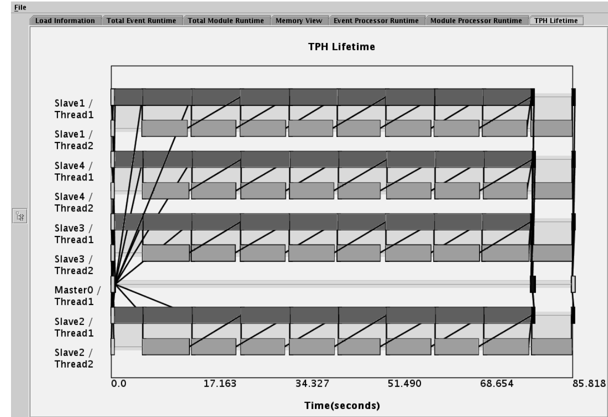

Figure 10. Synthetic Application Multi-threaded

load of modules is important for determining under which threads modules should run and if the application will even benefit from multithreading. With careful balance, on this architecture Coven can provide a performance improvement to an application of about 25%.

5.2. 2D Fast Fourier Transform

For a second application the 2D Fast Fourier Transform (FFT) was chosen. The 2D-FFT is used in a many applications and is often considered representative of workloads that operate on matrices that are distributed across the nodes of a parallel machine.

In an additional effort to demonstrate the simplicity of porting an existing application to Coven, an existing FFT library was linked in to Coven and a series of modules were created using it to perform a 2D-FFT. The Fastest Fourier Transform in the West (FFTW)[6] was developed by MIT and is considered one of the fastest FFT implementations available. The FFTW is open source and portable, unlike many vendor implementations. Complex double-precision floating-point numbers are used in the version considered here.

This algorithm is composed of the following four steps:

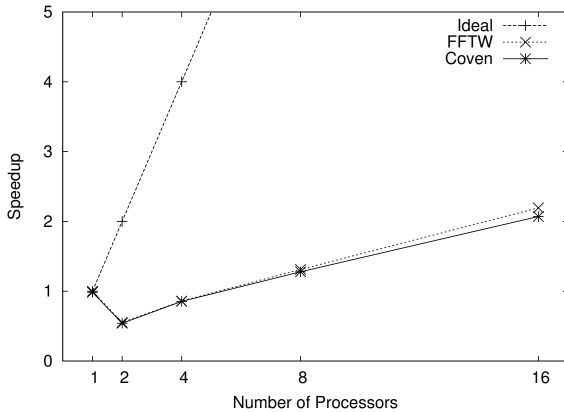

Figure 11. FFT Speedup - Coven vs. FFTW

- compute the 1D-FFT for each row
- transpose the matrix (redistribution of data)
- compute the 1D-FFT for each row
- transpose the matrix (redistribution of data)

These steps were translated directly into Coven modules, each containing no more than 3 lines of calls to FFTW libraries. The FFTW provides an MPI parallel implementation and this was used as a comparison application in these tests.

Firstly, a relatively small 4,000 x 4,000 2D-FFT was processed in parallel to compare the FFTW native implementation versus FFTW wrapped in Coven modules. Coven imposed an overhead between 1% to 5% of the total runtime for this application. Figure 11 shows the speedup curve for this application.

Next, a large 10,000 x 10,000 2D-FFT was considered. The resulting parallel portion of this FFT is too large to run on a small number of nodes on our system and so 8 and 16 nodes were chosen for testing. First, a single FFT was processed using FFTW and also a port of the application running in Coven. The results are shown in the first two rows of Table 2. Coven imposed an overhead of 6.6% in the 8 processor case and 5.1% in the 16 processor case.

Next, 3 FFTs of 10,000 x 10,000 size were processed. The timings for the single FFT case using FFTW was tripled to get the resulting time in row 3 of Table 2. Three separate instances of FFTW were then spawned concurrently in an attempt to let the operating system try and overlap computation and communication. The results are shown in row 4 of Table 2 which produce an improvement of between 9.3% and 13.3% over running 3 FFTs of this size back to back using FFTW.

Table 2. 10,000 x 10,000 Element FFT Timings

	8 Procs.	16 Procs.
1 FFT - FFTW	63.7s	42.9s
1 FFT - Coven	68.2s	45.2s
3 FFTs - FFTW		
Back-to-Back	191.1s	128.7s
Concurrent	173.2s	111.6s
Improvement	9.3%	13.3%
3 FFTs - Coven		
1 Thread	204.0s	137.1s
2 Threads	161.1s	105.2s
Improvement Over Back-to-Back FFTW	15.7%	18.2%
Improvement Over Concurrent FFTW	7.0%	5.7%

Finally, Coven was used to process the 3 FFTs. First, in the single threaded case row 6 of Table 2 shows that Coven performs more poorly than either approach using FFTW. However, when 2 threads were used in Coven, rows 7 and 8 of Table 2 show an improvement of between 15.7% and 18.2% over the back-to-back execution of 3 FFTs in parallel and between 5.7% and 7.0% over concurrent execution of 3 FFTs using FFTW.

Concurrently running multiple FFTW MPI parallel programs seems like a simple solution to process multiple FFTs but it has some drawbacks. For instance, while the operating system does an acceptable job of scheduling 3 FFTs concurrently, it is unlikely to handle 10 with the same efficiency. Furthermore, insufficient memory prob-

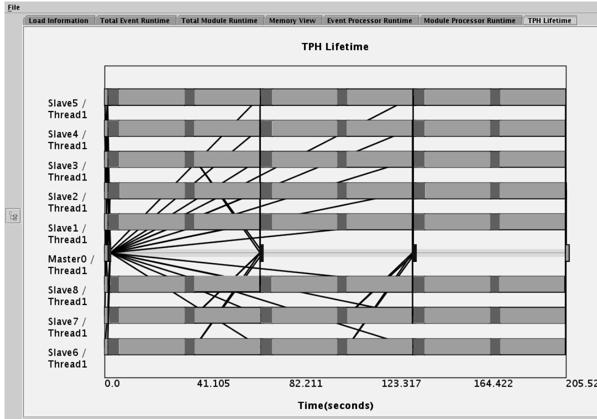

Figure 12. FFT Application Single Threaded

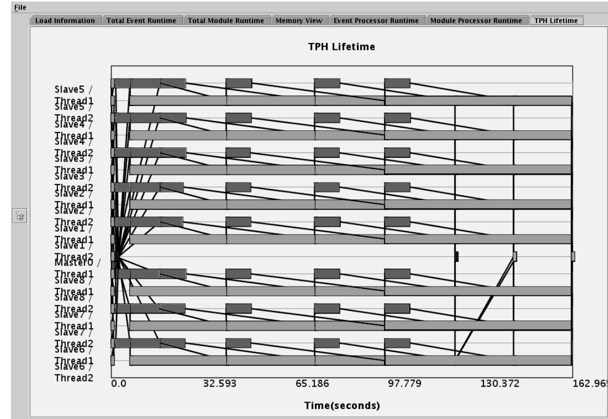

Figure 13. FFT Application Multithreaded

lems are likely to surface with this many large FFTs. Coven, on the other hand, uses a queue system described in Section 2.2. With this approach, the user can set it so that no more than 3 FFTs are being processed concurrently. Therefore, as one exits the system and the memory associated with it is freed, Coven can begin processing another FFT. The FFTW MPI implementation could be augmented to do something similar, however once an application is written in Coven, the programmer gets these features automatically.

The FFTW Coven application did not perform as well as the synthetic application. The reason for this is that the computation phase for this size 2D-FFT takes 5 seconds, while the communication phase takes nearly 30 seconds. This 1 to 6 ratio is no nearly as balanced as was the synthetic application. It is this reason that the FFTW Coven application does not approach the higher performance increases that are possible when using Coven's multithreaded capabilities.

Figure 12 contains a screenshot of the profiler for the 8 processor, single threaded version of this Coven application. The small blocked regions are the computation modules while the much wider regions are the communication modules.

Figure 13 contains a screenshot of this application running in multithreaded mode under Coven on 8 processors. After the first FFTs computation module, the FFT is handed to the second thread which begins a parallel transpose. The transpose operation on the multithreaded version takes between 33 and 34 seconds which is around 10% longer than the single threaded version. However, during this time the other two FFTs complete their 1-D computation. This overlap allows the application to achieve the performance gain described above.

Through careful scheduling of which modules execute concurrently, the multithreaded Coven version of the FFTW application was able to perform better than the naive execution of multiple FFTW MPI programs concurrently. Rather than have 3 FFTs competing for the CPU to perform computations concurrently, Coven lets one FFT get to the communication phase before it lets another into the computation phase.

6. Conclusions and Future Work

Many improvements have been made to Coven. Some improvements help the programmer by making applications easier to develop and debug while others make applications perform better. Multithreading parallel programs can lead to improved performance but requires applications which contain portions of code which release the CPU while waiting on an operation to complete. Network communication and disk I/O are two examples of such operations. If these phases are suf-

ficiently lengthy so that another task can accomplish some work concurrently, then performance benefits can be seen.

It was shown that a performance gain of around 25% can be achieved using Coven's multithreading capabilities if the application contains a balance between modules that load the CPU and ones that utilize the processor considerably less while waiting on an operation to complete. A powerful FFT library was easily wrapped in Coven modules and achieved between a 5.7% and 18.2% improvement in performance.

Problem solving environments and runtime systems generally add some sort of overhead to applications implemented in them over a conventionally coded application. However, if the system can provide optimizations or features which are easy to use yet would be complex for the programmer to implement in every application, then more application developers would seek to use these sorts of environments.

In the future, additional optimizations will be looked at to see if Coven can transparently provide a way for programmers in this environment to gain increased parallel program performance.

7. Acknowledgments

This work was supported by the ERC Program of the National Science Foundation under Award Number EEC-9731680. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

References

[1] Anthony Chan and William Gropp and Ewing Lusk. User's Guide for MPE: Extensions for MPI Programs. http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeman/mpeman.htm.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, pages 115–124, 1999.

[3] J. de St. Germain, J. McCorquodale, S. Parker, and C. Johnson. Uintah: A massively parallel problem solving environment. In *Ninth IEEE International Symposium on High Performance and Distributed Computing*, pages 33–41. IEEE, Piscataway, NJ, Nov 2000.

[4] N. A. DeBardeleben, W. B. Ligon III, S. Pandit, and D. C. Stanzione Jr. Coven - a framework for high performance problem solving environments. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002)*, pages 291–298, Edinburgh, Scotland, UK, July 2002. IEEE Computer Society.

[5] N. A. DeBardeleben, W. B. Ligon III, and D. C. Stanzione Jr. The Component-based Environment for Remote Sensing. In *Proceedings of the 2002 IEEE Aerospace Conference*, pages 6–2661–6–2670, March 2002.

[6] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[8] C. Johnson, S. Parker, D. Weinstein, and S. Heffernan. Component-based problem solving environments for large-scale scientific computing. *Journal on Concurrency and Computation: Practice and Experience*, (14):1337–1349, 2002.

[9] MCS Division, Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2.

[10] J. Rasure and S. Kubica. *The Khoros Application Development Environment*. Khoral Research Inc., Albuquerque, New Mexico, 1992.

[11] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From

trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.