Network of Workstations Active Messages Target for Ptolemy C Code Generation

by Patrick Warner

Memorandum No. UCB/ERL M97/8
January 24, 1997

Submitted to the Department of Electrical Engineering and Computer Science, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Abstract:

Ptolemy is a software environment for simulating, prototyping, and synthesizing heterogeneous systems. For studying network, parallel, and distributed systems, a Ptolemy Code Generation in C (CGC) domain target using Active Messages (AM) on a Network of Workstations (NOW) is a valuable tool. Active messages provide a low-overhead, portable communications protocol that can take advantage of performance improvements in the underlying network. A NOW provides a parallel platform to study Ptolemy simulations on. This project implements a NOW AM target for Ptolemy's CGC domain and explores the issues involved in creating appropriate NOW Ptolemy systems.

1. Introduction

Ptolemy is a software environment for simulating, prototyping, and synthesizing heterogeneous systems[1]. Examples of such systems include an ATM network for transmitting video signals, a Unix workstation and Digital Signal Processing (DSP) target board used for music tone generation, and a VHDL (VHSIC Hardware Description Language) model of a computer Central Processing Unit interacting with serial data. Ptolemy has existing support for network, parallel, and distributed systems through its use of dataflow modeling and scheduling. Ptolemy can model a system with dataflow paradigms, then generate a schedule that will execute the system in a parallel or concurrent manner. What is lacking in Ptolemy is a parallel platform on which to run the systems. The ability to run Ptolemy systems on a parallel platform, such as Berkeley's Network of Workstations (NOW) or any group of well networked workstations, would be valuable because the systems would be observed on an actual parallel system (as opposed to running a parallel simulation on a single workstation), and simulations would take less time to run because of the improved processing speed.

A NOW is a collection of stand-alone computers connected by a high-speed, low-latency, reliable network. A NOW can serve as a parallel platform for computing massive programs. The idea behind a NOW is to eventually create a distributed supercomputer on a building-wide scale. By using a low-latency, reliable, fully-connected network model, a NOW interconnect mirrors that of a massively parallel processor (MPP) computer. As Ptolemy schedulers can already map systems on to multiple processors, using a NOW with Ptolemy becomes a matter of defining how the processors will communicate with each other. A communication protocol matching that of a MPP seems reasonable for a NOW, and AM is a proven MPP communication method[2].

AM was originally developed strictly for MPP platforms as the Generic Active Messages (GAM) specification [3]. The GAM library was restricted to SPMD (Single Process Multiple Data) programs [4] that required the same code image to execute on each computing node. As the uses for GAM extended into more distributed, client-server computing (operating systems, file systems, network RAM), the SPMD model became too restrictive. The resulting AM specification eliminated the SPMD requirement and added an error model that had not existed in GAM. The Ptolemy multiprocessor code generation facilities automatically produce a different code image for each processor in a distributed system, making AM a more useful communication protocol on NOW than GAM. The AM implementation on Berkeley's NOW seeks to provide low-level access to the network hardware, an essential feature for effective parallel simulations.

With all the components necessary for NOW interaction with Ptolemy in place, the issue becomes how best to use these systems together. As mentioned briefly above, Ptolemy has existing support for parallel and distributed platforms in its schedulers and code generation facilities. Ptolemy is divided into models of computation called domains. Ptolemy's Code Generation (CG) domain takes modeled systems and translates them into a program. The Ptolemy Code Generation in C (CGC) domain generates C programs. The AM library is coded in C, so the CGC domain is an appropriate place to integrate the NOW platform. A Target within the CGC domain defines how the C code will be generated for a specific hardware platform. The solution for using Ptolemy and a NOW together is a NOW AM (NOWam) target in the CGC domain.

The main motivation in creating a CGC NOWam target is to expand the uses of Ptolemy into broader parallel and networking areas of research. With a parallel and distributed platform such as a NOW to run on, parallel and distributed Ptolemy simulations can be observed executing on an distributed system. The primary issue in implementing a CGC NOWam target is interprocess communication (IPC). IPC is accomplished using the AM library, so the AM library's implementation itself becomes an important issue in CGC NOWam target development.

A reference implementation of AM over the User Datagram Protocol/Internet Protocol (UDP/IP) (UDPAM) was created to test the AM specification. Although using UDP/IP doesn't conform to AM's goal of direct network hardware access, it does provide a useful and portable reference for testing purposes. Because of UDP/IPs computation overhead, bulk data transfers are more efficient than a fine grain message passing approach common to MPP platforms. At the time of this project, UDPAM is the only AM implementation available, so a bulk transfer technique for passing data is chosen.

The second part of this report discusses AM in more detail. Section 3 gives an overview of Ptolemy, focusing on code generation details. The following part of the report details the implementation of the NOWam target. Section 5 discusses parallel simulations for the NOWam target, and the results of running simulations on a NOW are discussed in Section 6. The final section summarizes the findings of the project and proposes future work.

2. Active Messages

AM seek to minimize the latency associated with communication protocol overhead and maximize user-level application bandwidth. Research at the University of California at Berkeley observed “communication protocols such as TCP/IP add unnecessary overhead to the base hardware cost”[5]. As network hardware continues to advance into the area of multi-gigabit bandwidths and sub-microsecond switch latencies, the existing protocol software overheads will dominate communication costs. AM provide a simple, portable, and general-purpose communications interface with direct application-network interactions that bypass the operating system on high-performance implementations [2].

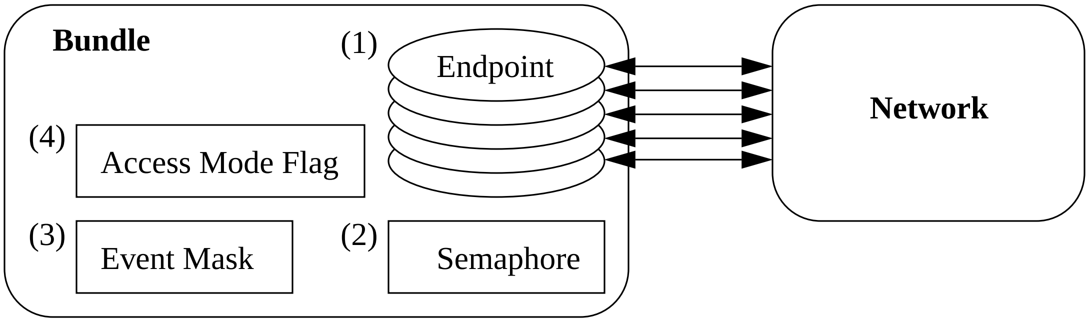
AM have been shown to be a useful means for building high-performance communication protocols, run-time environments, and message passing libraries on MPPs. In the original implementation of AM, GAM, all communication occurred within individual parallel processes with one network port per process. The parallel program itself had to be SPMD, a restriction making client/server applications and more general communication impossible. The current AM specification is no longer restricted to parallel SPMD applications, and is general enough to support such applications as file systems, operating systems, client/server programs, peer/peer programs, and parallel programs as well.

High-performance network hardware often uses an embedded processor or controller on the interface device, allowing communication and computation to overlap. This overlap feature is achieved by the embedded processor performing loads and stores from the communication host's memory. The operating system is bypassed in this case, and the user-level application has a direct connection to the network device using load and store operations. AM implementations seek to take advantage of the embedded processor to provide the direct application to network device channel.

The AM interface allows arbitrary serial and parallel processes to create multiple communication endpoints [2]. The communication endpoints are completely independent and secure network ports that resemble conventional Berkeley Unix sockets. Messages can be sent from any endpoint to any other endpoint, with the restriction that they use a common tag for protection purposes. Each endpoint has a message send pool, a message receive pool, a bulk transfer virtual memory segment, an authentication tag for receiving messages, a handler table that translates indices into handler functions, and a translation table that translates indices into global endpoint names.

A set of AM endpoints grouped together as a single unit for communication, synchronization, and event management is referred to as a bundle. The components of a bundle is shown in Figure 1. A bundle has (1) a collection of endpoints, (2) a thread synchronization variable, (3) an event mask that selects which state or state transitions generate events to an AM application, and (4) an access mode flag to indicate if concurrent access to the bundle or its endpoints is expected.

Figure 1: AM Bundle



The AM library is accessed through an ANSI C application programming interface (API), which is detailed in [2]. The features of primary interest to the NOWam target are:

`ea_t`
This type designates an endpoint address (see above for endpoint description).

`eb_t`

This type designates an endpoint bundle address (see above for bundle description).

```
handler_t
```

This type designates a handler-table index (described above with endpoints).

```
AM_Init()
```

This function initializes the AM layer and must be called before any interface function is used.

```
AM_Terminate()
```

This function cleans up and releases system resources used by AM and is called when finished using AM.

```
AM_Request4(
      ea_t request_endpoint, /* endpoint sending request */
      int reply_endpoint,   /* index of endpoint sending reply */
      handler_t handler,  /* index into destination endpoint’s handler table */
      int arg0, int arg1, int arg2, int arg3)
```

This function sends 4 integers to the destination. The receiver invokes the message handler when the request is received. The handler prototype is: void handler(void *token, int arg0, int arg1, int arg2, int arg3) where token represents the sending endpoint and arg0-3 are the 4 integer arguments.

```
AM_Reply4(
      void *token,
      handler_t handler, /* index into destination endpoint’s handler table */
      int arg0, int arg1, int arg2, int arg3)
```

This function sends the reply message to the requesting endpoint responsible for invoking the request handler making the reply. The receiver of the reply invokes a handler with the same prototype as `AM_Request4` above.

```
AM_RequestXferAsync4(
      ea_t request_endpoint, /* endpoint sending request */
      int reply_endpoint,   /* index of endpoint sending reply */
      int dest_offset,
      handler_t handler,  /* index into destination endpoint’s handler table */
      void *source_addr,
      int nbytes,
      int arg0, int arg1, int arg2, int arg3)
```

This function sends nbytes of contiguous data and then 4 integers to the destination. The receiver invokes the message handler with a pointer to the virtual-memory segment offset by dest_offset bytes, the number of data bytes transferred, and the 4 integers when the request is received. The handler prototype is: void handler(void *token, void *buf, int nbytes, int arg0, int arg1, int arg2, int arg3) where token represents the sending endpoint, buf is the virtual memory segment pointer with offset, nbytes is the number of bytes sent, and arg0-3 are the 4 integer arguments.

```
AM_Poll(eb_t bundle)
```

This function services incoming active messages from all endpoints in the specified bundle.

Communication between two endpoints is based on a request-reply model. An Active Message is sent from an endpoint send pool with an `AM_Request` or `AM_Reply`, and received into an endpoint receive pool. The message has an index that selects the handler function for the message from the receiving endpoint's handler table. After `AM_Poll` is called, the request or reply handler is invoked with the sent arguments. A bundle will handle multiple requests and replies on each `AM_Poll`. After messages are sent or received, they are removed from their respective message pools.

The above features will be used as the IPC mechanism in the NOWam target. With the IPC mechanism defined, the Ptolemy system itself, emphasizing the code generation facilities, must be examined to produce the NOWam target.

3. Ptolemy Overview

As stated in Section 1, Ptolemy is a software environment for simulating, prototyping, and synthesizing code for heterogeneous systems. Ptolemy uses an object-oriented framework (C++) to build its systems. A collection of various C++ classes are used to create Ptolemy applications. Using the base classes and paradigms of dataflow, higher-level constructs define Ptolemy's scheduling and code generation behavior. This section begins with a brief description of Ptolemy's base classes, and concludes with an introduction to its scheduling and code generation functionality.

The basic element of Ptolemy's modularity is the Block. The Block contains the go() method, which defines the Block's behavior at run-time. Derived from Block, the Star class is the elemental module in Ptolemy implemented by user code. A Star performs some computation, such as a single add or a complicated Fast Fourier Transform (FFT), or generates code to do so. A Galaxy, also derived from Block, contains a collection of Stars and/or other Galaxies internally. Another Block derivative, a Target, controls the execution of a Ptolemy application, or Universe. A Universe is a type of Galaxy that also inherits from class Runnable, which defines the execution and simulation or code generation behavior of the Universe.

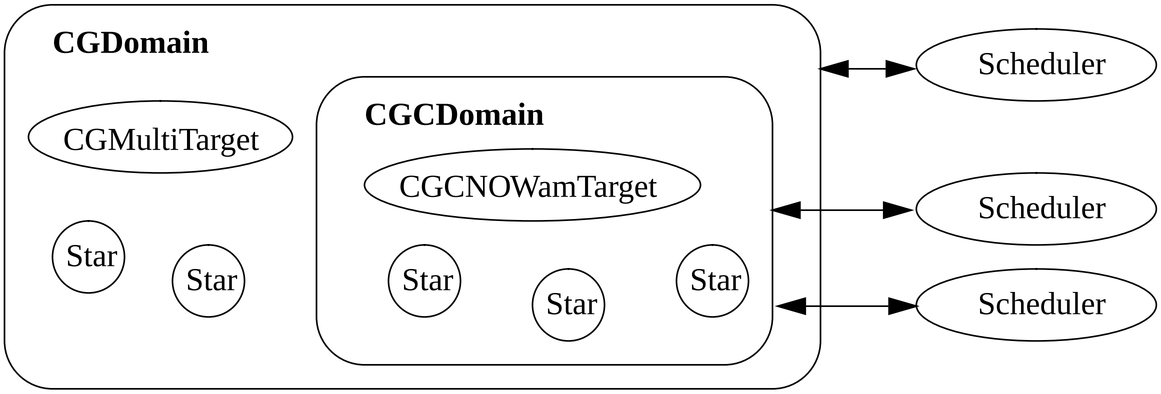
The model of computation for which Ptolemy code synthesis has been best defined is synchronous dataflow (SDF), a special case of the dataflow model of computation developed by Dennis [6]. SDF Stars produce and consume a constant number of data tokens at each invocation. Because of this, the execution order and resource requirements of SDF Stars can be determined at compile time. The SDF paradigm is used extensively in the definitions of Ptolemy's scheduling and code generation facilities.

With both a Universe of functional blocks to be scheduled and a Target describing the layout and behavior of the system for which code is to be generated, the Scheduler will: (a) determine which Block invocation will execute on which processor (for multi-processor systems); (b) determine the execution order of Stars on a processor; and (c) arrange the Stars' execution order into standard control structures, such as nested loops. After the scheduling phase, each processing element (single or multiple) is assigned scheduler-determined order of Blocks to execute. The Target class supplies the Scheduler with IPC information, enabling scheduling and code synthesis. The Scheduler uses IPC communication cost data to determine whether the cost is low enough to merit parallel execution.

The SDF model produces a graph that exposes the functional parallelism available in an algorithm. The next step in multiprocessor scheduling is to construct an Acyclic Precedence Expansion Graph (APEG) from the original SDF graph[7]. The APEG adds additional information such as communication costs between graph nodes and node computation costs. With the application APEG, a Ptolemy multiprocessor scheduler will map the graph nodes onto processors, taking into account IPC costs and communication overlap capabilities of the Target

A Ptolemy Domain is made up of Blocks, Targets, and Schedulers that follow a common computational model. A "computational model" in this context is the operational semantics that govern Block interaction. In code generation, a Domain's Blocks and Targets synthesize the same language. A Scheduler uses a Domain's definition to help order Block execution. Domains with the same computational model are subdivided further according to the language used. The CGC domain, under which the NOWam target is created, is derived from the CG domain. The CG domain obeys SDF semantics, allowing it to be scheduled at compile time. The relationship between the CG Domain, CGC Subdomain and their Targets, Schedulers, and Stars is pictured in Figure 2.

Figure 2: Ptolemy Domain and Subdomain Relationships



Code generation consists of two phases, scheduling and synthesis. The scheduling phase, described briefly above, partitions functional operators for possible parallel execution and orders Block execution on each target processor (single or multiple). Send and Receive Stars are spliced into the partition graph for IPC. In the synthesis phase, the scheduler-ordered code segments associated with each Block are stitched together. Most code generation work related to memory allocation and symbol generation is processor-independent, allowing these facilities to be contained in generic CG classes.

For a single processor CG Target, two scheduling choices are available, non-looping and looping. The first option, non-looping, is a quick and efficient scheduling method for applications that don't make abundant rate changes. The graph nodes are ordered according to Target resource costs. If Blocks are invoked many times, the generated code can become massive, as each invocation of a Block results in its own piece of code being added to the overall program. Using looping can result in dramatic code size reduction when the application makes large sample rate changes.

Multiple invocations of a Block can be placed inside a loop, with appropriate rate changes taking place between iterations.

Multiprocessor targets use the APEG described earlier to map graph nodes onto multiple processors. If IPC is ignored Hu's level-based list scheduling is used. This method is known a Highest Levels First with Estimated Times list scheduling, and assigns nodes to processors when they are ready to be executed and a processor is available, not taking communication cost into account. Using IPC communication costs and allowing overlapping communication, Sih's dynamic level (DL) scheduling[7] results in less communication because a node must have a high enough execution time to warrant the IPC overhead costs of transferring data between processors. This technique allows communication and computation to overlap, assuming dedicated communication hardware is available. The final Scheduler, Sih's declustering scheduling[7], makes multiple iterations over the graph, whereas Sih's DL scheduling makes a single pass, grouping nodes into "clusters" and analyzing trade-offs between parallel execution and IPC overhead.

The base CG star class (CGStar) contains the methods shared by all code generation Stars. CGStar consists of portholes, states, code blocks, a start() method, an initCode() method, a go() method, a wrapup() method and an execTime() method. Portholes designate the inputs and outputs of the Star. States represent parameters that can be set by the user or internal memory states needed in the generated code. Code blocks contain the target language and Star macro functions. Macro functions include parameter value substitution, unique symbol generation with multiple scopes, and state reference substitution [8]. The start() method is invoked prior to any scheduling or memory allocation because it initializes any information that will affect these actions. The initCode() method is called before scheduling but after memory allocation. The go() method is called by the Scheduler, synthesizing the code of the main loop. The wrapup() method places code outside of the main loop, after scheduling is done. The execTime() method returns a number that estimates the time to complete one Star invocation. This information is used by the Scheduler for determining the parallel execution of Stars within a Universe.

In review, Ptolemy applications, Universes, are made up of:
(a) elemental functions, Stars;
(b) collections of these functions and other collections, Galaxies;
(c) a scheduling procedure, Scheduler;
and (d) an execution host definition, Target.
A Universe is developed within a Domain, which describes how all the Universe's pieces interact with one another. For a CG Domain, the Stars eventually produce the Domain's target language, using their go() method. The Target definition designates how the code will be compiled and executed, and how IPC will take place. The Scheduler communicates with the Stars (execTime) and the Target (IPC mechanism) to determine the order of execution of all modules and any parallel execution possible. After the scheduling is completed, the code is synthesized by the CGStar methods, start(), initCode(), go(), and wrapup(). With the AM library described in section 2, and Ptolemy's code generation facilities outlined here, a CGC NOWam Target can be implemented.

4. NOW AM CGC Target

In order to create a new CGC target in Ptolemy two major components are needed: the target definition and the IPC mechanism. The target definition itself describes how the code will be collected, specifies and allocates resources, defines any needed platform initialization code, and finally dictates how to compile and run the generated code. The IPC component consists of send and receive actors for implementation.

The first step in creating a new target is to create a new instantiation of the Ptolemy Target class. In Ptolemy development, new Stars are usually created using a ptlang file. Ptlang is a high-level language which is preprocessed into C++ .h and .cc files for integration into the Ptolemy kernel. Ptlang cannot be used for defining CG targets, so C++ must be coded directly. The existing CGC target, CGCMultiTarget, was used as a model to code the CGCNOWamTarget from. A major difference in the resulting code is that CGCNOWamTarget inherits directly from the CGMultiTarget, which models a fully-connected multiprocessor architecture that allows overlapping communication. CGCMultiTarget inherits from CGSharedBus, which being based on a shared bus architecture, does not allow communication to overlap. The Berkeley NOW behaves as a fully-connected architecture, so the CGMultiTarget design is the best suited for it. The .h and .cc source files for the CGCNOWamTarget can be found in Appendix A. Because CGCNOWamTarget is a multiprocessor target, several additional design issues in Ptolemy are raised.

To support multiprocessor targets, a concept of parent-child target relationships is used [9]. The parent target defines the IPC mechanism and resources to be shared by the children. A hierarchy of child targets, which may themselves be complex heterogeneous multiprocessors or a single processor, completes the multiprocessor target definition. The child targets manage resources local to themselves. For the NOWam target, CGCNOWamTarget is the parent, and each processor(child) is represented by a default-CGC or Makefile-C target.

Sih’s DL scheduling is a natural match for the CGCNOWamTarget. IPC costs are considered in assigning APEG nodes to processors and DL scheduling allows communication and computation to overlap. This technique fits in well with the AM practice of using dedicated network hardware. Sih’s DL scheduling makes a single-pass over the graph, taking IPC overheads and resource constraints into account to schedule the communications and computations. IPC communication costs can be adjusted to have higher values for bulk transfers, and lowered when fast, dedicated network hardware is present.

As discussed in Section 3, the Scheduler splices Send and Receive Stars into the code where IPC takes place. It it up to the parent target in a multiprocessor target to define the Send and Receive stars. The Send and Receive Stars dictate the underlying communication mechanism used in the Target. For the CGCNOWamTarget, AM is used for the Send and Receive Stars.

The initial implementation of the NOWam target used a fine grain transfer approach, passing single floating point values between processors. This is a strait-forward approach that integrates well with the existing Ptolemy CG IPC architecture. At the time of this research UDPAM is the only AM implementation available. With UDPAM, a bulk transfer method is more efficient because the UDP/IP overhead is balanced out by sending large amounts of data with each UDP invocation. Ptolemy does not currently support a matrix data type in the CGC domain, so a conditional go() method was used. The code to send the data is inserted only once for each Send/Receive pair.

The Send star uses the AM_RequestXferAsync4() function described in Section 2, to send its data. AM_RequestXferAsync4() was used in lieu of AM_RequestXfer4 so that computation could overlap the communication. AM_RequestXferAsync4() returns control to the calling process immediately. The Receive Star includes the definition of the request handler function, and a call to AM_Poll(). The request handler moves the received data from the endpoint virtual memory segment into a local data structure that the rest of the application can access. The AM_Wait() function was initially used to block the Receive Star until new data arrived. Because each AM_Poll() serves multiple endpoints within an AM bundle, the Send and Receive Stars in an application would become unsynchronized when multiple Sends occurred before a Receive. To correct this problem, a synchronizing while loop was used in the Receive star. This causes the Receive to wait until its new data has arrived. Both the Send and Receive Stars were coded in ptlang, Ptolemy's high-level description language. After being preprocessed, .h and .cc files were produced for integration into the Ptolemy kernel. The .pl files for the CGCNOWamSend and CGCNOWamReceive can be examined in Appendix B.

The data being passed in the CGCNOWamTarget applications is double floating point numbers. For the original CGCNOWam target, a method was devised to pass single floating point values efficiently. This technique will be important for future CGCNOWam implementations where fine grain communication is acceptable, such as on the Myrinet LANai cards. There is no AM_Request message that directly sends double floating point data. For this reason, an initial IPC implementation used the AM_RequestI4() function, which sends a buffer of data. The double floating point number was sent as a stream of bytes and retrieved using the ANSI C memcpy() function. An improvement in performance can be achieved in AM by using the AM_Request4 function, which sends the least amount of data (4 integers) in AM. A C union type was created to transport a double floating point number as two integers. The type definition is:

```
typedef union ints_or_double {
         int asInt[2];
         double asDouble;
} convert;
```

The data is sent as follows:

```
convert myData;
myData.asDouble = double_float;
AM_Request4(endpoint, reply_index, handler_index,myData.asInt[0],
myData.asInt[1], 0, 0);
```

The data is retrieved as follows:

```
void handler(void *token, int arg0, int arg1, int arg2, int arg3)
{
     convert temp;
     temp.asInt[0] = arg0;
     temp.asInt[1] = arg1;
     Receive_double = temp.asDouble;
}
```

As UDPAM is the only AM implementation available at the time of this research, a bulk transfer method for sending data proves to be more efficient. The UPD/IP overhead results in simulations over 10 times slower running in parallel than on a single workstation when passing single floating point values. For the bulk transfer method, the data to be sent is first copied into the sending endpoint's virtual memory segment. `AM_RequestXferAsync()` is then invoked. On the receiving side, the data is copied out of the virtual memory segment. A possible performance enhancement would be to map the input and output data directly into the endpoint's virtual memory segments using `AM_SetSeg()`, eliminating two potentially large data copies. The disadvantage to this approach is that the local data segments might be corrupted by the application while the transfer is taking place asynchronously. For this reason, the copies have been left in place.

UDPAM uses IP address/port pairs to create the unique AM global endpoint names. The CGCNOWamTarget takes a list of host names corresponding to the number of processors, or network nodes, to be used in the Ptolemy application. The IP addresses of all the hosts are passed to the Send and Receive Stars to be used by the AM endpoint and bundle initialization code. Each Send/Receive pair in the application is represented by two endpoints with the same port number. Each child target in the CGCNOWamTarget has one bundle to handle all endpoint communication, synchronization, and event management. Multiple bundles were tested, but found to cause too much application overhead (each bundle creates a new thread to handle endpoint management).

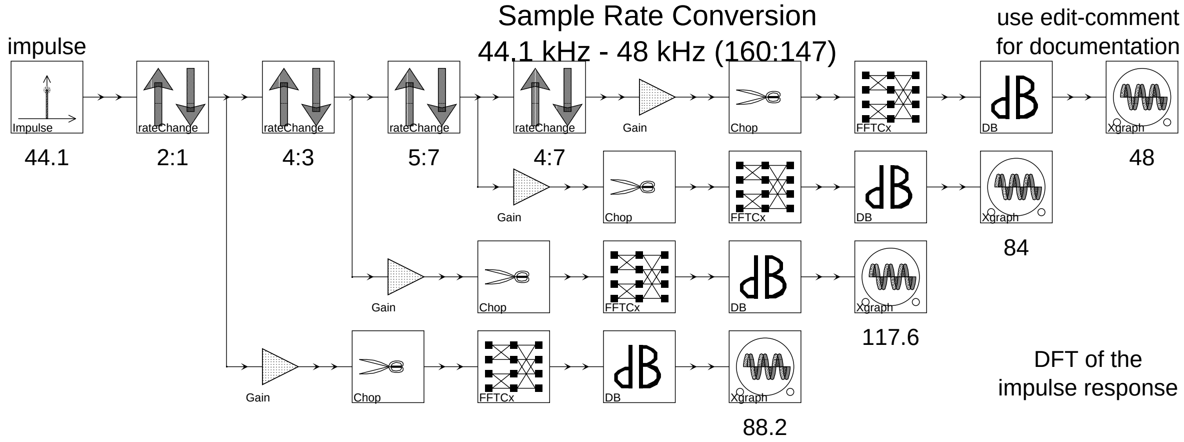
The AM specification does not supply any mechanism for sharing endpoint names across applications. This capability is not included because global endpoint names will differ from implementation to implementation (an IP address/port pair for UDPAM, a processor number for an MPP). It recommends the use of an external name server that stores endpoint names and supplies them to applications requesting them. In order to eliminate the need for an external name server in the CGCNOWamTarget, the UDPAM library was modified to use predefined ports in creating endpoints. This allowed the CGCNOWamTarget definition of global endpoint names as IP address/port pairs to be implemented without a name server.

With the CGCNOWamTarget implementation complete, appropriate Ptolemy simulations must be developed to test its correctness and performance.

5. Parallel Simulations

Appropriate Ptolemy applications for a NOW are those where computation dominates communication. In order to ensure that the CGCNOWamTarget produced correct results, an existing Ptolemy application with known results is desirable for testing. There are several existing Ptolemy simulations developed to model DSP systems with large amounts of computation. FFTs and Finite Impulse Response(FIR) filters involve performing complex math computations on potentially large amounts of data. One such simulation, an up-sample system, accomplishes the difficult task of converting signal sampling rates from 44.1 kHz to 48 kHz. The rate conversion is performed in multiple stages, each stage of which can be mapped on to multiple processors. See Appendix C for a detailed description of what the system does. The schematic is pictured in Figure 3.

Figure 3: Up Sample Ptolemy Schematic

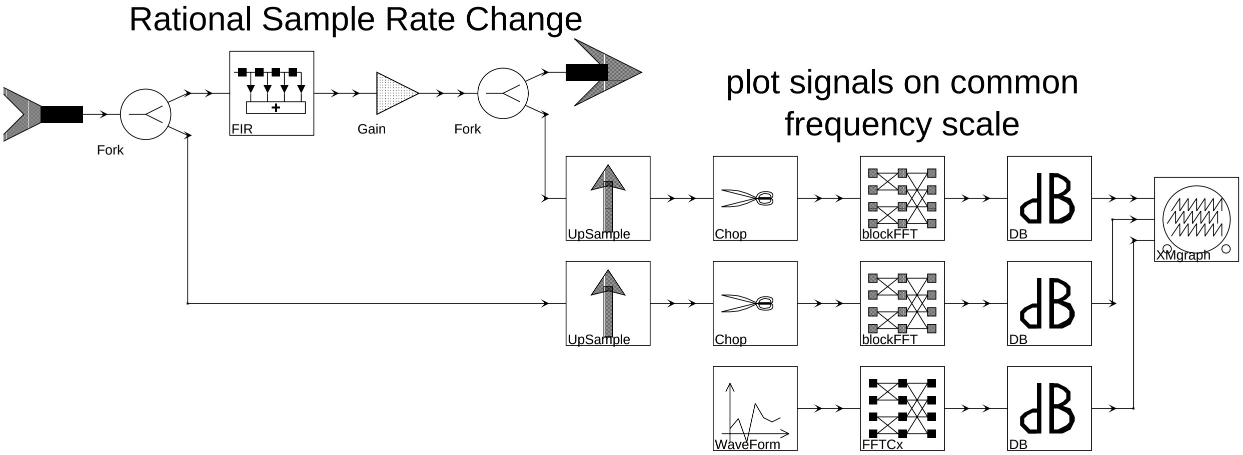


Manual scheduling was used for this simulation in order to create well-defined divisions for bulk transfers. The simulation was tested with only one bulk transfer taking place between processor 1 and processor 2, one between processor 2 and processor 3, and one between processor 3 and processor 4. Sending all the data at once is anticipated to make the UDP/IP overhead insignificant.

Alternate schedules were also produced in order to compare the different scheduling algorithms. Although Ptolemy's built-in schedulers were able to produce schedules in most cases, the computer's memory was exhausted before code could be generated. Because of the lack of source code to compile and test, the respective scheduler's executions could not be compared, but their schedules could (see the next section, Results).

When manual scheduling is used, Ptolemy follows a one Star on one processor rule. This limits parallelism that may be available by spreading Stars across multiple processors. Each of four processors was assigned one rate Galaxy to execute, and all Stars leading up to one of the respective graph Stars, labeled with frequency values on the schematic pictured in Figure 3. The internals of the rate Galaxy are pictured in Figure 4.

Figure 4: Rate Ptolemy Schematic



The work in each of the rate Galaxies is not exactly equal due to the different sampling rates. Each processor does execute one FIR filter and four FFTs, but these computations differ in the number of data samples operated on and in the order of the FFTs. In looking at the schedule, an equal number of FFTs with equal execution times appear on each processor. The difference in workload is most obvious in counting the number of FIR filters executed by each processor. Processor 1 executes 147 FIR filters, processor 2 executes 98 FIR filters, processor 3 executes 56 FIR filters, and processor 4 executes 40 FIR filters. These differences are due to the different sampling rates of each rate Galaxy. Refer to Appendix C for more information on this simulation. Examining the information from the schedule, processor 1 is observed to do the most amount of work. A benefit of the up-sample system is that all of the processors can overlap computation immediately. Within each of the rate Galaxies, a wave signal is passed into an FFT. Before the last 3 processors wait to receive data, they can work on this part of the simulation while the first processor prepares to send the data. Unlike the other processors, processor 1 does not have to wait for data to arrive to start computing. This fact may help balance out processor 1's heavier workload.

For a two processor test, the first two stages of the up-sample simulation were used. The simulation was also scheduled manually, in order to provide a better comparison with the four processor simulation. Because of the two processor test's smaller size, the built-in Ptolemy schedulers may have been able to generate code. But the manual schedule makes the overall experiment more uniform, so it is used. The same division for processor 1 and 2 is used for the two processor test, where one bulk transfer is made between processor 1 and 2. An attempt was made to schedule all four stages of the up-sample simulation on just two processors, but the generated code was too large and exhausted the computer's memory during compilation.

A side-effect of using manual scheduling was non-looping code being generated for each processor. As discussed in Section 3, non-looping code leads to simulations with changing sampling rates to generating code for every invocation of a Star, resulting in massive code explosion. As an example, the WaveForm Star inside the rate Galaxy is invoked 256 times. Rather than a loop with 256 iterations, the generated code contains 256 instances of the WaveForm code. This code explosion is what caused the compiler to run out of memory when compiling all four stages of the up-sample simulation on two processors. A solution to this problem is to use Ptolemy's hierarchical scheduling[10]. This scheduler allows one of the three multiprocessor schedulers discussed in Section 3 to be used as a top-level scheduler. For each child of the multiprocessor target, a single processor looping scheduler is used to reduce code explosion. The effects of using this scheduling technique is left to future research.

In order to determine the maximum speedup possible for the simulation, information from the Ptolemy schedule can be used. Speedup is defined as the time it takes the program to execute on a single processor divided by the total elapsed time to execute the program in parallel[4]. A graph with node computation costs and communication costs is pictured in Figure 5. Each node in the graph is assigned to a processor, taking into account communication delays and delays resulting from scheduling. The scheduling delays result when a node must wait for another node to complete before it can compute. The schedule, including communication delay, is shown in Figure 6. The time to execute the program in serial is calculated by adding up the node computation times, 2 + 1 + 2 + 4 + 2 + 2 + 2 + 2 + 4 = 21. The parallel time is the total time over the breadth of the graph, or makespan, 10. The speedup is calculated by taking the time to execute the program in serial divided by the makespan, 21 / 10 = 2.1.

Figure 5: Example APEG

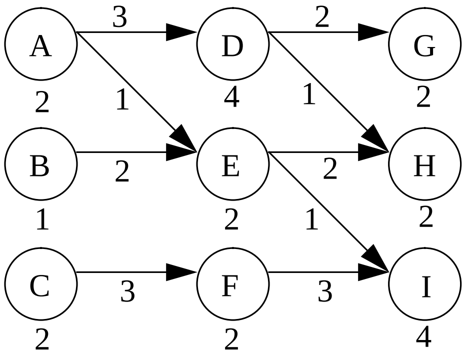
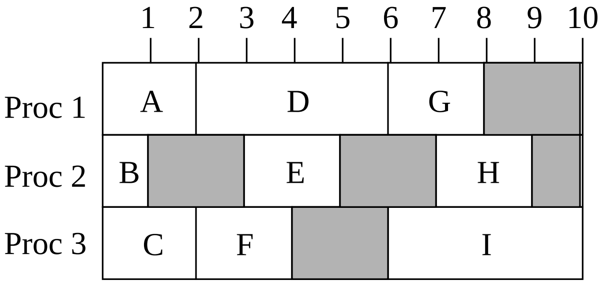


Figure 6: Example Multiprocessor Schedule



Another way to calculate the maximum speedup is to determine how much of the parallel time all processors are being used, reported the Ptolemy schedule as the utilization. For the schedule in Figure 6, processor 1 has a 8 / 10 = 80% utilization, processor 2 has a 5 / 10 = 50% utilization, and processor 3 has a 8 / 10 = 80% utilization. The total for all 3 processors is (80% + 50% + 80%) / 3 = 70%. Without any communication or scheduling delays, the maximum speedup would be equal to the number of processors, 3. With communication and scheduling delays accounted for, the maximum speedup is .7 x 3 = 2.1.

6. Results

The use of manual scheduling for the multiprocessor simulations resulted in non-looping code being generated. For a fair comparison, the single workstation code was initially generated using a non-looping schedule, which resulted in a huge C source file (191736 lines). This code was not able to be compiled because the computer's memory was exhausted. Instead, a looping schedule was used for the single processor code, which resulted in a 65 times reduction in code size (2967 lines). Timing measurements were made, and the results for the two and four processor simulations are presented below.

The resulting schedule for four processors reports that 74% utilization is achieved, showing that the processors are computing 74% of the total execution time, and idle for 26% of the time. Taking this into account, a maximum speedup of .74 x 4 = 2.96 is theoretically possible for the four processor simulation.

The two processor simulation schedule shows 84% utilization, signifying a 1.68 maximum speedup should be possible.

For the two processor simulation, two SparcStation 20s with 128 MB of RAM connected by 100 Mbps switched Ethernet network were used. The results are summarized in Table 1. As can be seen, the simulation attains approximately a 1.6 times speedup running on two processors over one. The speedup is calculated using the time of processor 2 in this case, as both processors begin computing at the same time. Processor 2 spends 17% of its time, on average, waiting for the data from processor 1 to arrive. The time is takes processor 1 to send data, about 1.3 ms on average, is negligible. If the time waiting for the receive data could be eliminated, the maximum speedup of 1.68 would be approached. Efficiency in parallel computing is the speedup divided by the number of processors[4]. For the two processor simulation, the efficiency is 80%.

Table 1: Two Processor Results

	Processor 1	Processor 2	Single Processor
Send Time	1.310 ms	N/A	N/A
Receive Time	N/A	138.8 ms	N/A
Execution Time	696.0 ms	827.2 ms	1316.0 ms

The four processor simulation was carried out using three SparcStation 20s with 128 MB of RAM, and one SparcStation 10 with 80 MB of RAM, all connected by a 100 Mbps switched Ethernet network. The results are summarized in Table 2. The timing for the stand-alone system differed from the SparcStation 20s to the SparcStation 10. The SparcStation 20s were about 1.7 times faster than the SparcStation 10 in executing the program on a single processor. The average time between all four of these systems was used to calculate the one processor value. The fourth processor in this simulation spends almost 50% of its time waiting for its data to arrive. The third processor spends about 23% of its total time waiting for data, the second processor almost 10%. As in the two processor system, removing these times would allow the maximum speedup to be approached. This waiting time is due to scheduling. The latter processors finish the work that is not dependent on the data they receive before the receive data is ready to be sent. This results in the processors wasting compute cycles waiting for that data to arrive. This wait time increases with later processors, as in this simulation all the processors do the same amount of computation before waiting for receive data to arrive. More efficient scheduling could undoubtedly improve performance, but better scheduling in this specific simulation may not be applicable as there is only a limited amount of work that can be carried out without the received data (essentially one FFT). The speedup for the four processor simulation is 1.8, the efficiency is 45%.

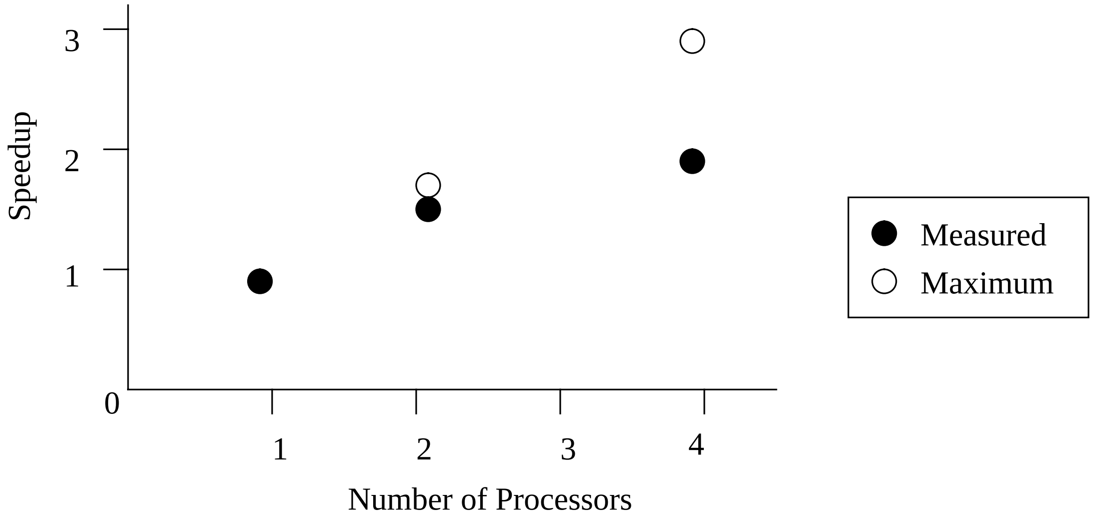
Table 2: Four Processor Results

	Processor 1	Processor 2	Processor 3	Processor 4	Single Processor
Send Time	1.140 ms	1.230 ms	0.712 ms	N/A	N/A
Receive Time	N/A	98.10 ms	287.9 ms	782.8 ms	N/A
Execution Time	804.8 ms	976.3 ms	1234.1 ms	1647.9 ms	2884.3 ms

An alternate schedule that divides the rate Galaxy work more evenly among the processors may result in better performance, but this would also incur more communication costs. A schedule was generated using DL scheduling and assuming fine grain communication. This schedule resulted in 72% utilization with communication sends and receives were spread throughout the generated code. With the bulk transfer method, the DL scheduler arranged sends and receives into larger groupings, and a 75% utilization was reported.

These results show that a definite benefit is gained in using multiple processors (workstations) for large Ptolemy simulations. With better scheduling, higher efficiencies can be achieved. When the times are measured in milliseconds, the benefits may not appear as significant, but milliseconds are very significant to a CPU. The speedup is graphed in Figure 7.

Figure 7: Up-Sample Simulation Speedup



7. Summary

An initial implementation of a CGCNOWamTarget was built using GAM. The target was not functional due to GAM’s requirement of a SPMD parallel program, i.e. each node in the process had to be running the same code image. Test data was collected by stitching each file produced by the CGCNOWamTarget into a single file that was then compiled and run on each node. The GAM implementation tested was built on TCP/IP, so the TCP/IP overhead canceled any benefit gained using Berkeley’s NOW.

The UDPAM implementation of the CGCNOWamTarget is fully integrated into Ptolemy, as the SPMD requirement of GAM is no longer part of the AM specification. Because UDPAM is built on top of UDP/IP, a CGCNOWamTarget application can be created, compiled, and run any network supporting UDP/IP. The use of UDP/IP is a benefit in the sense CGCNOWamTarget can be run on systems other than Berkeley’s NOW, but it has the same detriment that the original CGCNOWamTarget using TCP/IP did, costly UDP/IP overhead. Even with protocol overhead, appropriate Ptolemy applications can be designed and run on Berkeley’s NOW or other networks supporting UDP/IP. These applications would simply have to have a higher degree of computation than communication.

The up-sample simulations show that Ptolemy systems do exist that benefit from running on multiple processors. The 80% efficiency and 1.6 times speedup achieved with two processors is excellent and certainly shows an added benefit from running the simulation on two processors rather than one. These values were close the maximum possible speedup of 1.68 and efficiency of 84%. Although the simulation appears not to scale well to four processors, the 1.8 times speedup is amplified in importance if overall execution time is critical, and is 60% of the possible maximum of 3. If a massive simulation took 10 days to run on one processor, only taking 5 1/2 days would definitely prove beneficial, even at only 45% efficiency. If nothing else, these results show the potential for creating Ptolemy simulations that will execute in less time on multiple processors, and the CGCNOWam target is a means of using multiple processors.

The current CGCNOWamTarget is not portable across AM implementations because of its use of known ports in creating endpoints. Although a name server is suggested in the AM documentation, no interface is provided. If a name server was developed for the CGCNOWamTarget, there is still no guarantee the name server interface would be the same for other AM implementations. Although not restricting global names of endpoints is an essential feature of the AM specification for portability, it is a potential weakness unless a common name server interface is defined.

Although the building-wide supercomputer has not yet been realized, the CGCNOWam target running over 100 Mbps Ethernet shows the goal is being approached. It is already the case where accessing data over the network from the memory of another computer is faster than accessing a local hard disk. This fast network communication lends itself naturally to using networked computers as a parallel platform.

Future potential work on the CGCNOWamTarget includes a port to the new LAM II library, which is an AM implementation running on Myrinet LANai network cards. This implementation is true to the AM goals in that it provides a direct link between an application and the network hardware, bypassing the operating system. This new CGCNOWamTarget would perform closely to a MPP system, and would be able to run a wider range of Ptolemy applications more efficiently than the UDPAM CGCNOWamTarget. Other future work could include experimenting with additional Ptolemy scheduling techniques and developing more NOW simulations.

8. References

[1] Joseph Buck, Soonhoi Ha, Edward A. Lee, David G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” International Journal of Computer Simulation, 1992.

[2] Alan M. Mainwaring, “Active Message Applications Programming Interface and Communication Subsystem Organization,” Draft Technical Report, University of California at Berkeley, Computer Science Department, 1996.

[3] David Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, Chad Yoshikawa, “Generic Active Message Interface Specification,” Version 1.1, University of California at Berkeley, Computer Science Department, 1995.

[4] Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[5] Remzi H. Arpaci, Andrea Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson, “The Interaction of Parallel and Sequential Workloads on a Network of Workstations,” Technical Report CS-94-838, University of California at Berkeley, Computer Science Department, 1994.

[6] Edward A. Lee and David G. Messerschmitt, “Synchronous Data Flow,” Proceedings of the IEEE, September, 1987.

[7] Gilbert C. Sih, “Multiprocessor Scheduling To Account For Interprocessor Communication”, Ph.D. Thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, 1991.

[8] Jose L. Pino, “Software Synthesis for Single-Processor DSP Systems Using Ptolemy,” Master’s Report, University of California at Berkeley, Department of Electrical Engineering and Computer Science, 1994.

[9] Jose L. Pino, Soonhoi Ha, Edward A. Lee, Joseph T. Buck, “Software Synthesis for DSP Using Ptolemy,” Journal of VLSI Signal Processing, 9, 7-21, 1995.

[10] Jose L. Pino and Edward A. Lee, “Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors,” Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, May 1995, pp. 2643-2646.

Appendix A

CGCNOWamTarget.h

```
/******************************************************************
Version identification:
@(#)CGCNOWamTarget.h1.6 3/4/96

Copyright (c) 1995-1996  The Regents of the University of California.
All Rights Reserved.

Permission is hereby granted, without written agreement and without
license or royalty fees, to use, copy, modify, and distribute this
software and its documentation for any purpose, provided that the above
copyright notice and the following two paragraphs appear in all copies
of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
PROVIDED HEREUNDER IS ON AN “AS IS” BASIS, AND THE UNIVERSITY OF
CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.
                                                        COPYRIGHTENDKEY

 Programmer: Patrick Warner

*******************************************************************/

#ifndef _CGCNOWamTarget_h
#define  _CGCNOWamTarget_h 1

#ifdef __GNUG__
#pragma interface
#endif

#include “CGMultiTarget.h”
#include “StringState.h”
#include “IntArrayState.h”
#include “IntState.h”

class EventHorizon;
class CGCTarget;

class VirtualInfo {
friend class CGCNOWamTarget;
      unsigned long inetAddr; // internet address
      int virtNode;           // active message virtual node
      const char* nm;         // machine name
public:
      VirtualInfo(): virtNode(0), nm(0) {}
};

class CGCNOWamTarget : public CGMultiTarget {
public:
      CGCNOWamTarget(const char* name, const char* starclass, const char*
desc);
      ~CGCNOWamTarget();

      Block* makeNew() const;
      int isA(const char*) const;

      // redefine IPC funcs
      DataFlowStar* createSend(int from, int to, int num);
      DataFlowStar* createReceive(int from, int to, int num);

      // spread and collect
      DataFlowStar* createSpread();
      DataFlowStar* createCollect();

      // redefine
      void pairSendReceive(DataFlowStar* s, DataFlowStar* r);

      // get VirtualInfo
      VirtualInfo* getVirtualInfo() { return machineInfo; }
      void setMachineAddr(CGStar*, CGStar*);

      // signal TRUE when replication begins, or FALSE when ends
      void signalCopy(int flag) { replicateFlag = flag; }

protected:
      void setup();

      // redefine
      int sendWormData(PortHole&);
      int receiveWormData(PortHole&);

private:
      // states indicate which machines to use.
      StringState machineNames;
      StringState nameSuffix;

      // In case, the cody body is replicated as in “For” and “Recur”
      // construct, save this information to be used in getMachineAddr().
      IntArray* mapArray;
      int baseNum;
      int replicateFlag;

      // information on the machines
      VirtualInfo* machineInfo;

      // number of send/receive pairs
      int pairs;

      // identify machines
      int identifyMachines();

      // return the machine_id of the given target.
      int machineId(Target*);
};

#endif
```

CGCNOWamTarget.cc

```
static const char file_id[] = “CGCNOWamTarget.cc”;
/******************************************************************
Version identification:
@(#)CGCNOWamTarget.cc1.10 3/8/96

Copyright (c) 1995-1996  The Regents of the University of California.
All Rights Reserved.

Permission is hereby granted, without written agreement and without
license or royalty fees, to use, copy, modify, and distribute this
software and its documentation for any purpose, provided that the above
copyright notice and the following two paragraphs appear in all copies
of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
PROVIDED HEREUNDER IS ON AN “AS IS” BASIS, AND THE UNIVERSITY OF
CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.
                                                        COPYRIGHTENDKEY

 Programmer: Patrick Warner

*******************************************************************/


#ifdef __GNUG__
#pragma implementation
#endif

#include “pt_fstream.h”
#include “Error.h”
#include “CGUtilities.h”
#include “CGCStar.h”
```

```
#include “KnownTarget.h”
#include “CGCNOWamTarget.h”
#include “CGCTarget.h”
#include “CGCSpread.h”
#include “CGCCollect.h”
#include “CGCNOWamSend.h”
#include “CGCNOWamRecv.h”
#include <ctype.h>
#include <stdio.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>  // Sol2 needs this for inet_addr()

// stream for logging information.  It is opened by the setup method.

static pt_ofstream feedback;

// -----------------------------------------------------------------------------
CGCNOWamTarget::CGCNOWamTarget(const char* name,const char* starclass,
               const char* desc) : CGMultiTarget (name,starclass,desc) {

      // specify machine names
      addState(machineNames.setState(“machineNames”,this,”lucky, babbage”,
“machine names (separated by a comma)”));
      addState(nameSuffix.setState(“nameSuffix”,this,””,
            “common suffix of machine names(e.g. .berkeley.edu)”));

      // make some states invisible
      childType.setInitValue(“default-CGC”);
      compileFlag.setInitValue(“NO”);
      runFlag.setInitValue(“NO”);
      displayFlag.setInitValue(“YES”);
      resources.setInitValue(““);

      machineInfo = 0;
      pairs = 0;
      baseNum = 0;
      mapArray = 0;
      replicateFlag = 0;
}

CGCNOWamTarget :: ~CGCNOWamTarget() {
//    if (inherited() == 0) {
            LOG_DEL; delete [] machineInfo;
//    }
}

// -----------------------------------------------------------------------------
DataFlowStar* CGCNOWamTarget :: createSend(int, int, int) {
      LOG_NEW; CGCNOWamSend* s = new CGCNOWamSend;
      return s;
}
```

```
DataFlowStar* CGCNOWamTarget :: createReceive(int, int, int) {
      LOG_NEW; CGCNOWamRecv* r =  new CGCNOWamRecv;
      return r;
}

DataFlowStar* CGCNOWamTarget :: createSpread() {
      LOG_NEW; return (new CGCSpread);
}

DataFlowStar* CGCNOWamTarget :: createCollect() {
      LOG_NEW; return (new CGCCollect);
}

void CGCNOWamTarget :: pairSendReceive(DataFlowStar* s, DataFlowStar* r) {
      feedback << “\tpairing “ << s->fullName() << “ --> “ << r->fullName()
            << “\n”; feedback.flush();
      CGCNOWamSend* cs = (CGCNOWamSend*) s;
      CGCNOWamRecv* cr = (CGCNOWamRecv*) r;
      int pnum = (int)nprocs;

      cs->numNodes.setInitValue(pnum);
      cr->numNodes.setInitValue(pnum);

      cs->pairNumber.setInitValue(pairs);
      cr->pairNumber.setInitValue(pairs);
      pairs++;

      StringList nodeAddrs = “ “;
      for (int i = 0; i < pnum; i++) {
            nodeAddrs << (int)(machineInfo[i].inetAddr) << “ “;
      }
      cs->nodeIPs.setInitValue(hashstring(nodeAddrs));
      cr->nodeIPs.setInitValue(hashstring(nodeAddrs));

      cs->partner = cr;
}

void CGCNOWamTarget :: setMachineAddr(CGStar* s, CGStar* r) {
      CGCNOWamSend* cs = (CGCNOWamSend*) s;
      CGCNOWamRecv* cr = (CGCNOWamRecv*) r;

      CGTarget* tg = cr->cgTarget();
      if (replicateFlag && mapArray) {
            int six = -1;
            int rix = -1;
            CGTarget* sg = cs->cgTarget();
            int numMatch = 0;
            for (int i = 0; i < mapArray->size(); i++) {
                  CGTarget* temp = (CGTarget*) child(mapArray->elem(i));
                  if (temp == sg) {
                        six = i; numMatch++;
                  } else if (temp == tg) {
                        rix = i; numMatch++;
                  }
                  if (numMatch >= 2) break;
            }
            if ((six < 0) || (rix < 0) || (numMatch != 2)) {
                  Error :: abortRun(“setMachineAddr failed.”);
                  return;
            }
            int zz = six /baseNum;
            if ((rix / baseNum) != zz) {
                  int newIx = zz * baseNum + (rix % baseNum);
                  tg = (CGTarget*) child(mapArray->elem(newIx));
            }
      }

      // machine address
      int dix = machineId(tg);
      if (dix < 0) {
            Error :: abortRun(*cr, “no child target for this star.”);
            return;
      }
      cs->hostAddr.setInitValue(machineInfo[dix].virtNode);
}

int CGCNOWamTarget :: machineId(Target* t) {
      for (int i = 0; i < nChildrenAlloc; i++) {
            if (child(i) == t) return i;
      }
      return -1;
}

// ---------------------------------------------------------------------------
                  ///////////////////
                  // setup
                  ///////////////////

void CGCNOWamTarget :: setup() {
//    if (inherited()) {
//          CGCNOWamTarget* orgT = (CGCNOWamTarget*) child(0)->parent();
//          machineInfo = orgT->getVirtualInfo();
//          CGMultiTarget :: setup();
//          return;
//    }

      // all runs will append to the same file.
      // FIXME: should not be done this way.
      if (!feedback) feedback.open(“CGCNOWam_log”);
      if (!feedback) return;

      // machine idetifications
      if (identifyMachines() == FALSE) return;

      CGMultiTarget :: setup();

      // machine name setup
      for (int i = 0; i < nChildrenAlloc; i++) {
```

```
                CGCTarget* t = (CGCTarget*) child(i);
                t->setHostName(machineInfo[i].nm);
        }
        feedback.flush();
}

int CGCNOWamTarget :: identifyMachines() {
        // construct machine information table
        int pnum = int(nprocs);
        LOG_NEW; machineInfo = new VirtualInfo[pnum];

        feedback << “ ** machine identification ** \n”;
        const char* p = machineNames;
        int i = 0;
        while (*p) {
                char buf[80], *b = buf;
                while (isspace(*p)) p++;
                while ((*p != ‘,’) && (*p != 0)) {
                        if (isspace(*p)) p++;
                        else *b++ = *p++;
                }
                if (*p == ‘,’) p++;
                *b = 0;         // end of string.
                // record names
                StringList mname = (const char*) buf;
                mname << (const char*) nameSuffix;
                machineInfo[i].nm = hashstring((const char*) mname);
                // internet address calculation
                struct hostent* hp;
                if ((hp = gethostbyname((const char*) mname)) == NULL) {
                        StringList errMsg;
                        errMsg << “host name error: “ << mname;
                        Error :: abortRun(errMsg);
                        return FALSE;
                }
                struct in_addr* ptr = (struct in_addr*) hp->h_addr_list[0];
                machineInfo[i].inetAddr = inet_addr(hashstring(inet_ntoa(*ptr)));
                machineInfo[i].virtNode = i;

                // monitoring.
                feedback << “machine(“ << i << “) = “;
                feedback << mname << “: “;
                feedback << machineInfo[i].virtNode << “\n”;
                feedback.flush();

                i++;
        }

        // check if the number of processors and the machine names are matched.
        if (i != pnum) {
                Error :: abortRun(*this, “The number of processors and”,
                        “ the number of machine names are not equal.”);
                return FALSE;
        }
```

```
      return TRUE;
}

// ---------------------------------------------------------------------------
Block* CGCNOWamTarget :: makeNew() const {
      LOG_NEW; return new CGCNOWamTarget(name(),starType(),descriptor());
}
// ---------------------------------------------------------------------------
                  /////////////////////////////
                  // wormhole interface method
                  /////////////////////////////

int CGCNOWamTarget :: receiveWormData(PortHole& p) {
      CGPortHole& cp = *(CGPortHole*)&p;
      cp.forceSendData();
      return TRUE;
}
// ---------------------------------------------------------------------------
int CGCNOWamTarget :: sendWormData(PortHole& p) {
      CGPortHole& cp = *(CGPortHole*)&p;
      cp.forceGrabData();
      return TRUE;
}

// ---------------------------------------------------------------------------
ISA_FUNC(CGCNOWamTarget,CGMultiTarget);

static CGCNOWamTarget targ(“CGCNOWam”,”CGCStar”,
“A NOW target for parallel C code generation”);

static KnownTarget entry(targ,”CGCNOWam”);
```

Appendix B

CGCNOWamRecv.pl

```
defstar {
      name { NOWamRecv }
      domain { CGC }
      desc {
Receive star between NOW processors.
      }
      version { @(#)CGCNOWamRecv.pl1.25 8/22/96 }
      author { Patrick Warner }
      copyright {
Copyright(c) 1995-1996 The Regents of the University of California
All rights reserved.
See the file $PTOLEMY/copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
}
      location { CGC NOW Active Message target library }
      explanation {
Produce code for inter-process communication (receive-side).
      }
      private {
            friend class CGCNOWamTarget;
      }
      output {
            name {output}
            type {FLOAT}
      }

      state {
            name { numData }
            type { int }
            default { 1 }
            desc { number of tokens to be transferred }
            attributes { A_NONSETTABLE }
      }
        state {
                name { nodeIPs }
                type { intarray }
                default { “0 1 2 3” }
                desc { IP addresses of nodes in program. }
                attributes { A_NONSETTABLE }
        }
        state {
                name { numNodes }
                type { int }
                default { 0 }
                desc { Number of nodes in program. }
                attributes { A_NONSETTABLE }
        }
        state {
                name { pairNumber }
                type { int }
                default { 0 }
                desc { Send Receive pair number for unique IP port. }
                attributes { A_NONSETTABLE }
        }
        state {
                name { runCount }
                type { int }
                default { 0 }
                attributes { A_NONSETTABLE }
        }
        defstate {
                name { localData }
                type { floatarray }
                default { “0” }
                attributes { A_NONSETTABLE }
        }

      setup {
                numData = 400;
                localData.resize(numData);
                output.setSDFParams(int(numData), int(numData)-1);
      }

      codeblock (outData) {
        for (i = $val(numData) - 1; i >= 0; i--) {
                $ref(output,i) = $ref(localData)[j++];
        }
      }
      codeblock (timeincludes) {
#ifdef TIME_INFO1
#include <sys/time.h>
#endif
      }
      codeblock (ipcHandler) {
void $starSymbol(ipc_handler)(void *source_token, void *buf, int nbytes,
                              int d1, int d2, int d3, int d4)
{
        $starSymbol(RecvData) = 100.0;
}
      }
        codeblock (errorHandler) {
void error_handler(int status, op_t opcode, void *argblock)
{
        switch (opcode) {
                case EBADARGS:
                        fprintf(stderr,”Bad Args:”);
                        fflush(stderr);
                        break;
                case EBADENTRY:
                        fprintf(stderr,”Bad Entry:”);
                        fflush(stderr);
                        break;
                case EBADTAG:
                        fprintf(stderr,”Bad Tag:”);
                        fflush(stderr);
                        break;
                case EBADHANDLER:
                        fprintf(stderr,”Bad Handler:”);
                        fflush(stderr);
                        break;
                case EBADSEGOFF:
                        fprintf(stderr,”Bad Seg offset:”);
                        fflush(stderr);
                        break;
                case EBADLENGTH:
                        fprintf(stderr,”Bad Length:”);
                        fflush(stderr);
                        break;
                case EBADENDPOINT:
                        fprintf(stderr,”Bad Endpoint:”);
                        fflush(stderr);
                        break;
                case ECONGESTION:
                        fprintf(stderr,”Congestion:”);
                        fflush(stderr);
                        break;
                case EUNREACHABLE:
                        fprintf(stderr,”Unreachable:”);
                        fflush(stderr);
                        break;
        }
}
        }
        codeblock (amdecls) {
en_t global;
eb_t bundle;
        }
      codeblock (timedecls) {
#ifdef TIME_INFO1
hrtime_t timeRun;
hrtime_t beginRun;
hrtime_t endRun;
#endif
      }
      codeblock (stardecls) {
#ifdef TIME_INFO3
hrtime_t $starSymbol(timeRecv);
hrtime_t $starSymbol(beginRecv);
hrtime_t $starSymbol(endRecv);
#endif
int $starSymbol(i);
en_t *$starSymbol(endname);
ea_t $starSymbol(endpoint);
      }
        codeblock (aminit) {
AM_Init();
if (AM_AllocateBundle(AM_PAR, &bundle) != AM_OK) {
        fprintf(stderr, “error: AM_AllocateBundle failed\n”);
        exit(1);
}
if (AM_SetEventMask(bundle, AM_NOTEMPTY) != AM_OK) {
        fprintf(stderr, “error: AM_SetEventMask error\n”);
        exit(1);
}

        }
      codeblock (timeinit) {
#ifdef TIME_INFO3
$starSymbol(timeRecv) = 0.0;
#endif
#ifdef TIME_INFO1
beginRun = gethrtime();
#endif
      }
      codeblock (starinit) {
$starSymbol(RecvData) = -0.001;

if (AM_AllocateKnownEndpoint(bundle, &$starSymbol(endpoint),
&$starSymbol(endname), HARDPORT + $val(pairNumber)) != AM_OK) {
        fprintf(stderr, “error: AM_AllocateKnownEndpoint failed\n”);
        exit(1);
}

if (AM_SetTag($starSymbol(endpoint), 1234) != AM_OK) {
        fprintf(stderr, “error: AM_SetTag failed\n”);
        exit(1);
}
if (AM_SetHandler($starSymbol(endpoint), 0, error_handler) != AM_OK) {
        fprintf(stderr, “error: AM_SetHandler failed\n”);
        exit(1);
}
if (AM_SetHandler($starSymbol(endpoint), 2, $starSymbol(ipc_handler)) !=
AM_OK) {
        fprintf(stderr, “error: AM_SetHandler failed\n”);
        exit(1);
}
if (AM_SetSeg($starSymbol(endpoint), (void *)$ref(localData), $val(numData) *
sizeof(double)) != AM_OK) {
        fprintf(stderr, “AM_SetSeg error\n”);
        exit(-1);
}
for ($starSymbol(i) = 0; $starSymbol(i) < $val(numNodes); $starSymbol(i)++) {
        global.ip_addr = $ref(nodeIPs, $starSymbol(i));
        global.port = HARDPORT + $val(pairNumber);
        if (AM_Map($starSymbol(endpoint), $starSymbol(i), global, 1234) !=
AM_OK) {
                fprintf(stderr, “AM_Map error\n”);
                fflush(stderr);
                exit(-1);
        }
}
      }
      initCode {
            addGlobal(“#define HARDPORT 61114\n”, “hardPort”);
            addGlobal(“double $starSymbol(RecvData);\n”);
            addInclude(“<stdio.h>”);
            addInclude(“<stdlib.h>”);
            addInclude(“<string.h>”);
            addInclude(“<thread.h>”);
            addInclude(“<udpam.h>”);
            addInclude(“<am.h>”);
            addCompileOption(
                  “-I$PTOLEMY/src/domains/cgc/targets/NOWam/libudpam”);
            addLinkOption(
                  “-L$PTOLEMY/lib.$PTARCH -ludpam -lnsl -lsocket -lthread”);

            addCode(timeincludes, “include”, “timeIncludes”);
            addProcedure(ipcHandler);
            addProcedure(errorHandler, “CGCNOWam_ErrorHandler”);
            addCode(amdecls, “mainDecls”, “amDecls”);
            addCode(timedecls, “mainDecls”, “timeDecls”);
            addCode(stardecls, “mainDecls”);
            addCode(timeinit, “mainInit”, “timeInit”);
            addCode(aminit, “mainInit”, “amInit”);
            addCode(starinit, “mainInit”);
      }

      codeblock (block2) {
      /* run receive once */
      }
      codeblock (block) {
      int i, j = 0;

#ifdef TIME_INFO3
      $starSymbol(beginRecv) = gethrtime();
#endif

      while ($starSymbol(RecvData) == -0.001) {
            if (AM_Poll(bundle) != AM_OK) {
                  fprintf(stderr, “error: AM_Poll failed\n”);
                  exit(1);
            }
      }
      $starSymbol(RecvData) = -0.001;

#ifdef TIME_INFO3
      $starSymbol(endRecv) = gethrtime();
      $starSymbol(timeRecv) += $starSymbol(endRecv) - $starSymbol(beginRecv);
      printf(“Cumulative time to receive %lld usec\n”, $starSymbol(timeRecv) /
1000);
#endif
      }

      go {
                if (runCount == 0) {
                  addCode(block);
                  addCode(outData);
                        runCount = 1;
                } else {
                        addCode(block2);
                }
      }
      codeblock (runtime) {
#ifdef TIME_INFO1
endRun = gethrtime();
timeRun = endRun - beginRun;
printf(“Time to run %lld usec\n”, timeRun / 1000);
#endif
      }
      wrapup {
            addCode(runtime, “mainClose”, “runTime”);
            addCode(“AM_Terminate();\n”, “mainClose”, “amTerminate”);
      }
}
```

CGCNOWamSend.pl

```
defstar {
      name { NOWamSend }
      domain { CGC }
      desc {
Send star between NOWam processors.
      }
      version { @(#)CGCNOWamSend.pl1.20 8/22/96 }
      author { Patrick O. Warner }
      copyright {
Copyright(c) 1995-1996 The Regents of the University of California
        }
      location { CGC NOW Active Message target library }
      explanation {
Produce code for inter-process communication (send-side)
      }
      private {
            friend class CGCNOWamTarget;
            CGStar* partner;
      }
      input {
            name {input}
            type {FLOAT}
      }
      state {
            name { numData }
            type { int }
            default { 1 }
            desc { number of tokens to be transferred }
            attributes { A_NONSETTABLE }
      }
        state {
                name { nodeIPs }
```

```
          type { intarray }
          default { “0 1 2 3” }
          desc { IP addresses of nodes in program. }
          attributes { A_NONSETTABLE }
  }
state {
      name { hostAddr }
      type { int }
      default { 0 }
      desc { Host virtual node for server }
      attributes { A_NONSETTABLE }
}
  state {
          name { numNodes }
          type { int }
          default { 0 }
          desc { Number of nodes in program. }
          attributes { A_NONSETTABLE }
  }
  state {
          name { pairNumber }
          type { int }
          default { 0 }
          desc { Send Receive pair number for unique IP port. }
          attributes { A_NONSETTABLE }
  }
  state {
          name { runCount }
          type { int }
          default { 0 }
          attributes { A_NONSETTABLE }
  }
  defstate {
          name { localData }
          type { floatarray }
          default { “0” }
          attributes { A_NONSETTABLE }
  }

hinclude { “CGCNOWamTarget.h” }

setup {
      numData = 400;
      localData.resize(numData);
      input.setSDFParams (int(numData), int(numData)-1);
}

  codeblock(loadCode) {
  int i, check, j = 0;
  for (i = $val(numData) - 1; i >= 0; i--) {
          $ref(localData)[j++] = $ref(input,i);
  }
  }
  codeblock (timeincludes) {
```

```
#ifdef TIME_INFO1
#include <sys/time.h>
#endif
        }
        codeblock (replyHandler) {
void reply_handler(void *source_token, int d1, int d2, int d3, int d4)
{
}
        }
        codeblock (errorHandler) {
void error_handler(int status, op_t opcode, void *argblock)
{
        switch (opcode) {
                case EBADARGS:
                        fprintf(stderr,”Bad Args:”);
                        fflush(stderr);
                        break;
                case EBADENTRY:
                        fprintf(stderr,”Bad Entry:”);
                        fflush(stderr);
                        break;
                case EBADTAG:
                        fprintf(stderr,”Bad Tag:”);
                        fflush(stderr);
                        break;
                case EBADHANDLER:
                        fprintf(stderr,”Bad Handler:”);
                        fflush(stderr);
                        break;
                case EBADSEGOFF:
                        fprintf(stderr,”Bad Seg offset:”);
                        fflush(stderr);
                        break;
                case EBADLENGTH:
                        fprintf(stderr,”Bad Length:”);
                        fflush(stderr);
                        break;
                case EBADENDPOINT:
                        fprintf(stderr,”Bad Endpoint:”);
                        fflush(stderr);
                        break;
                case ECONGESTION:
                        fprintf(stderr,”Congestion:”);
                        fflush(stderr);
                        break;
                case EUNREACHABLE:
                        fprintf(stderr,”Unreachable:”);
                        fflush(stderr);
                        break;
        }
}
        }
        codeblock (amdecls) {
en_t global;
```

```
eb_t bundle;
        }
        codeblock (timedecls) {
#ifdef TIME_INFO1
hrtime_t timeRun;
hrtime_t beginRun;
hrtime_t endRun;
#endif
        }
        codeblock (stardecls) {
#ifdef TIME_INFO2
hrtime_t $starSymbol(timeSend);
hrtime_t $starSymbol(beginSend);
hrtime_t $starSymbol(endSend);
#endif
en_t *$starSymbol(endname);
ea_t $starSymbol(endpoint);
int $starSymbol(i);
        }
        codeblock (timeinit) {
#ifdef TIME_INFO2
$starSymbol(timeSend) = 0.0;
#endif
#ifdef TIME_INFO1
beginRun = gethrtime();
#endif
      }
        codeblock (aminit) {
AM_Init();
if (AM_AllocateBundle(AM_PAR, &bundle) != AM_OK) {
        fprintf(stderr, “error: AM_AllocateBundle failed\n”);
        exit(1);
}
if (AM_SetEventMask(bundle, AM_NOTEMPTY) != AM_OK) {
        fprintf(stderr, “error: AM_SetEventMask error\n”);
        exit(1);
}

        }
        codeblock (starinit) {
if (AM_AllocateKnownEndpoint(bundle, &$starSymbol(endpoint),
&$starSymbol(endname), HARDPORT + $val(pairNumber)) != AM_OK) {

        fprintf(stderr, “error: AM_AllocateKnownEndpoint failed\n”);
        exit(1);
}

if (AM_SetTag($starSymbol(endpoint), 1234) != AM_OK) {
        fprintf(stderr, “error: AM_SetTag failed\n”);
        exit(1);
}
if (AM_SetHandler($starSymbol(endpoint), 0, error_handler) != AM_OK) {
        fprintf(stderr, “error: AM_SetHandler failed\n”);
        exit(1);
}
if (AM_SetHandler($starSymbol(endpoint), 1, reply_handler) != AM_OK) {
        fprintf(stderr, “error: AM_SetHandler failed\n”);
        exit(1);
}
if (AM_SetSeg($starSymbol(endpoint), (void *)$ref(localData), $val(numData) *
sizeof(double)) != AM_OK) {
      fprintf(stderr, “AM_SetSeg error\n”);
        exit(-1);
}
for ($starSymbol(i) = 0; $starSymbol(i) < $val(numNodes); $starSymbol(i)++) {
        global.ip_addr = $ref(nodeIPs, $starSymbol(i));
        global.port = HARDPORT + $val(pairNumber);
        if (AM_Map($starSymbol(endpoint), $starSymbol(i), global, 1234) !=
AM_OK) {
                fprintf(stderr, “AM_Map error\n”);
                fflush(stderr);
                exit(-1);
        }
}
        }

      initCode {
            // obtain the hostAddr state from parent MultiTarget.
            // Note that this routine should be placed here.
            CGCNOWamTarget* t = (CGCNOWamTarget*) cgTarget()->parent();
            t->setMachineAddr(this, partner);
            hostAddr.initialize();

            // code generation.
            addGlobal(“#define HARDPORT 61114\n”, “hardPort”);
            addInclude(“<stdio.h>”);
            addInclude(“<stdlib.h>”);
            addInclude(“<thread.h>”);
            addInclude(“<udpam.h>”);
            addInclude(“<am.h>”);
            addCompileOption(
                      “-I$PTOLEMY/src/domains/cgc/targets/NOWam/libudpam”);
            addLinkOption(
                 “-L$PTOLEMY/lib.$PTARCH -ludpam -lnsl -lsocket -lthread”);

            addCode(timeincludes, “include”, “timeIncludes”);
            addProcedure(replyHandler, “CGCNOWam_ReplyHandler”);
            addProcedure(errorHandler, “CGCNOWam_ErrorHandler”);
            addCode(amdecls, “mainDecls”, “amDecls”);
            addCode(timedecls, “mainDecls”, “timeDecls”);
            addCode(stardecls, “mainDecls”);
            addCode(timeinit, “mainInit”, “timeInit”);
            addCode(aminit, “mainInit”, “amInit”);
            addCode(starinit, “mainInit”);
      }

      codeblock (block2) {
      /* run send once */
      }
      codeblock (block) {

#ifdef TIME_INFO2
      $starSymbol(beginSend) = gethrtime();
#endif

      check = AM_RequestXferAsync4($starSymbol(endpoint), $val(hostAddr), 0,
2, (void *)$ref(localData), $val(numData)*sizeof(double), 0, 0, 0, 0);
      if (check == -1) {
            fprintf(stderr, “Error in sending data\n”);
            fflush(stderr);
      }

#ifdef TIME_INFO2
      $starSymbol(endSend) = gethrtime();
      $starSymbol(timeSend) += $starSymbol(endSend) - $starSymbol(beginSend);
      printf(“Cumulative time to send %lld usec\n”, $starSymbol(timeSend) /
1000);
#endif

      }
      go {
            if (runCount == 0) {
                  addCode(loadCode);
                  addCode(block);
                  runCount = 1;
            } else {
                  addCode(block2);
            }
      }

        codeblock (runtime) {
#ifdef TIME_INFO1
endRun = gethrtime();
timeRun = endRun - beginRun;
printf(“Time to run %lld usec\n”, timeRun / 1000);
#endif
        }

      wrapup {
                addCode(runtime, “mainClose”, “runTime”);
                addCode(“AM_Terminate();\n”, “mainClose”, “amTerminate”);
      }
}
```

Appendix C

Up-Sample Simulation (from Ptolemy Galaxy comments)

Converting sampling rates from 44.1 kHz to 48 kHz is a difficult problem. A naive approach would be to interpolate (upsample) to a sampling frequency which is the least common multiple of these two frequencies, filter to prevent aliasing, then decimate (downsample) to the desired output rate. Unfortunately the sampling rate ratio in this case is 160:147. This would require interpolating to an intermediate frequency of 7.056 MHz. Designing a lowpass filter with a pass band of 0-20 kHz and a stop band of 22.05-3528 kHz would be very challenging. Such a high-Q filter would require many, many coefficients to obtain reasonable performance.

A better approach is to perform the rate conversion in multiple stages. Rate conversion ratios are chosen by examining the prime factorization of the two sampling rates. The prime factorizations of 48000 and 44100 are 2^7 * 3 * 5^3 and 2^2 * 3^2 * 5^2 * 7^2, respectively. Thus the ratio 48000:44100 is 2^5 * 5 : 3 * 7^2 or 160:147. In this example the conversion is performed in four stages - 2:1, 4:3, 5:7, and 4:7.

The first stage requires a filter with a relatively sharp cut-off with a transition band from 20-22.05 kHz. Because of this, the ratio for this stage was chosen to be 2:1. With the smallest possible interpolation factor of 2, the cut-off frequency of 20 kHz is as high as possible with respect to the intermediate sampling rate (88.2 kHz in this case). This means that the filter for this stage will require fewer coefficients than if a higher interpolation factor had been chosen. Unfortunately, no decimation can take place in this stage since the smallest decimation factor is 3. The first filter, which has 173 taps, interpolates by a factor of 2 and does not decimate. The pass band is 0-20 kHz and the stop band is 22.05-44.1 kHz. Note that the filter operates at a sampling rate of 2x44.1 = 88.2 kHz. The output of this filter is a signal at a 88.2 kHz sampling rate with no energy above 22.05 kHz.

The second filter, which has 31 taps, interpolates by a factor of 4 and decimates by a factor of 3. The pass band is 0-20 kHz and the stop bands are 44.1 kHz wide and are centered at multiples of 88.2 kHz (the sampling rate of the input to this stage). More specifically, the stop bands are 66.15-110.25 kHz and 154.35-176.4 kHz. Note that this filter operates at a sampling rate of 4x88.2 = 352.8 kHz. The output of this filter is a signal at a 117.6 kHz sampling rate.

The third filter, which has 33 taps, interpolates by a factor of 5 and decimates by a factor of 7. The pass band is 0-20 kHz and the stop bands are 44.1 kHz wide and are centered at multiples of 117.6 kHz (the sampling rate of the input to this stage). More specifically, the stop bands are 95.55-139.65 kHz and 213.15-257.25 kHz. Note that this filter operates at a sampling rate of 5 x 117.6 = 588 kHz. The output of this filter is a signal at a 84 kHz sampling rate.

The fourth filter, which has 33 taps, interpolates by a factor of 4 and decimates by a factor of 7. The pass band is 0-20 kHz and the stop bands are 44.1 kHz wide and are centered at multiples of 84 kHz (the sampling rate of the input to this stage). More specifically, the stop bands are 61.95-106.05 kHz and 145.95-168 kHz. Note that this filter operates at a sampling rate of 4 x 84 = 336 kHz. The output of this filter is a signal at a 48 kHz sampling rate. Because the second filter has the same interpolation factor as the fourth and operates at a higher rate, it can actually use the same filter coefficients.