Code Generation for Process Network Models onto Parallel Architectures

Man-Kit Leung
Isaac Liu
Jia Zou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-139
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-139.html

October 28, 2008

Code Generation for Process Network Models onto Parallel Architectures

Man-kit Leung, Isaac Liu, and Jia Zou
Center for Hybrid and Embedded Software Systems, EECS
University of California, Berkeley
Berkeley, CA 94720, USA
{mankit, liuisaac, jiazou}@eecs.berkeley.edu

Abstract

With multi-core and many-core architectures becoming the current focus of research and development, and as vast varieties of architectures and programming models emerging in research, the design space for applications is becoming enormous. From the number of cores, the memory hierarchy, the interconnect to even the programming model and language used are all design choices that need to be optimized for applications in order to fully benefit from parallel architectures. We propose a code generation framework targeting rapid design space exploration and prototyping. From the high level design, code for specific architectures and mappings can be generated and used for comparison. We choose Khan Process Networks[11] as our current specification language, because of its inherit parallelism and expressiveness. Our code generator take advantage of Message Passing Interface (MPI) [6] as the API for implementing message passing across platforms. We show the scalability of the generated MPI code and the ability to extend our framework to allow for tuning and optimization.

1 Introduction

The shift from single-core sequential code to multi/many-core parallel code has not been as intuitive as one would have hoped. Simply running a program on a parallel architecture will not necessarily yield a performance increase. In some cases, it might even degrade performance due to the overhead created by the parallel architectures. Thus, efforts such as the Berkeley Parallel Computing Laboratory (ParLab) [1] have gathered scholars with different areas of expertise to help with the transition to parallel computing. From the underlying architecture to the parallelizing of applications, all levels of abstraction are being rethought and redeveloped. While the efforts and results of researchers and academia have been promising, the wide range of methods and solutions delivered create an enormous design space for end users.

Programming in parallel itself is already a daunting task. Much effort is required to insure correctness and prove the program to be dead-lock free, not to mention additional tuning and optimizing for performance. However, even before any programming can be done, the underlying architecture must be decided. From the number of cores, the memory hierarchy, to the inter-connection network used are all application specific parameters that need to be optimized. Choosing the right mix often requires extensive research and time, which leads to slower product development cycles. To allow more rapid prototyping and development, we build upon the design methodology of "Correct-by-Construction" proposed by Dijkstra [4]. Dijkstra states that if a series of mathematically-correct transformations are applied to a mathematically-correct model, then the resulting transformation is also mathematically-correct. In the same way, designers would first construct higher level models to ensure and prove correctness of their design. Then, transform the higher level model to actual implementation. That transformation is code generation.

We propose a code generation framework that generates parallel code targeting different platforms from higher level specifications which allow for quick development and prototyping of parallel applications. This framework will allow users to parametrize several design choices such as number of cores, targeting library, and partitioning of the application to quickly generate executable parallel code for comparison and tuning. We further extend this framework to insert profiling and feedback code into the generated program. This allows users to obtain execution trace and statistics, which can be fed back to the code generator to further tune and optimize the to produce better code. We implemented this code generation framework on top of the Ptolemy II project, which is a heterogeneous modeling and simulation environment designed to allow users to explore high level models of computations[3]. Currently, a MPI code generation engine has been implemented and able to generate

MPI code from Process Network models. Our results show low overhead when comparing to the current pthreads implementation used for Process Network models.

The following sections describe our work. First we give our work context in terms of other research in the same area. Then we will give some background information on the languages and framework we used. Following, we give an explanation of our code generation framework, including a work flow of our code generator. We further explain the implementation details of the generator, and finally conclude with some testing results and conclusion.

2 Related work

Prior work in [18] has been done to generate code for Active Messages(AM), which is a lower-level mechanism that can be used to implement data parallel or message passing efficiently. Due to the fact AM is a communication primitive, the functionalities supported by it is very limited compared to that of MPI, which is built upon AM. Thus one way to looked at it is that AM's functionality is a subset of that of MPI. Also in this work by Warner, the generated scheduler is of Synchronous Dataflow(SDF) semantics. SDF is a special case of Process Networks, where the firings of all actors could be scheduled at compile time. However being a special case of Process Networks, SDF also has less expressiveness, meaning that some models that could be modeled by Process Networks will not be modeled by SDF, but not the other way around.

Another work [14] also tries to do code generation for multiprocessor platforms. Like the last one, it also focuses on system modeled by SDF model of computation. Also, this work actually require a set of send and receive actors. This means every time a specific partition is made, these actors needs to be inserted. Our work does not have this restriction, where changing the partition does not require us to manually change the model itself. Rather, the communication between processors are indicated by port attributes.

[16] is also of important relevance to us. In this work, the authors still focused on SDF model of computation. SDF provides edge and node weights in a very delicate way, where a acyclic precedence graph could be constructed from the model, and the node and edge weights are practically given. However, one of the constraints is that the model builder must have information about the details of the platform in order to translate the number of firings of a particular actor in an "iteration" to node weights, and number of communications between two actors in an "iteration" to edge weights. However in our work, as we explain later, no model of the platform is needed.

StreamIt [17] also builds upon SDF, with a focus mainly on streaming applications. This had the implication that each actor in the system do not have states associated with them. Now this may not be much of a disadvantage since this project targets very fine grained parallelism with the system, thus functionalities such as FIR filtering could be performed by having actors in parallel, and having a delay in the system to delay the data, thus producing the effect of state storage. On the contrary, the implication of our work focuses on coarse grain parallelism, where each actor could have some states stored in them. This has a greater implications for all applications, where instead of exploring parallelism in streaming applications only, the expressiveness of our model could be much bigger. This also implies we could incorporate a large amount of legacy code that is already written, which helps in exploring parallel novel applications, where parts of implementations already exist.

Finally, Parks did a comparison study between MPI and Process Networks (PN) as programming models in [**?**]. PN is more restrictive in its communication semantics. A PN communication channel consists of strictly one producer and one consumer, which means an input channel may not receives from more than one data sources. It also requires every process to perform blocking read on all of its input data streams. On the other hand, MPI provides primitives that are more generic. He showed that the PN communication model is analogous to the blocking send and receive primitives of MPI; thus, any PN models can easily be emulated using MPI primitives. However, it is not possible to emulate every MPI primitive using the PN communication semantics. He demonstrated how to use to PN to model several of the more common MPI primitives (*broadcast, scatter, and gather*). While Parks compares PN and MPI as equivalent-class programming models, our work uses PN as the higher level programming model and take advantages of MPI as the run-time for our generated code.

3 Background

3.1 Kahn Process Networks

Process Networks (PN) is first propsed by Gilles Kahn [11] as a distributed model of computation where actors (logical processing blocks) are connected by communication channels. Every communication channel assumes an infinite-sized FIFO buffer queues. PN is intrinsically concurrent since every actor may execute its computation independent of other actors if given the availability of inputs. Each actor essentially executes in a separate thread of execution. Actors may issue blocking or nonblocking writes to the communication buffers but blocking reads is strictly required. Although the question of whether or not the execution of a given PN model may deadlock is undecideable, there is an determinate order for the data exchanged at every communication stream. The PN model of computation provides a very expressive programming model while retaining

determinism for a programmer.

3.2 Ptolemy & Code Generation

Ptolemy II is a graphical software system for modeling, simulation, and design of concurrent, real-time, embedded systems. It uses the notion of model of computation (MoC), also called domain, to describe the interaction between components. For example, PN is one of these MoCs implemented in Ptolemy II. Ptolemy II focuses on assembly of concurrent components with well-defined MoCs. Many features in Ptolemy II contribute to the ease of its use as a rapid prototyping environment. For example, domain polymorphism allows one to use the same component in multiple MoCs. Data polymorphism and type inference mechanisms automatically take care of type resolution, type checking and type conversion, and make users unaware of their existence most of the time. A rich expression language makes it easy to parameterize many aspects of a model statically or dynamically. However, these mechanisms add much indirection overhead and therefore cannot be used directly in an implementation.

The code generation framework takes a model that has been shown to meet certain design specifications through simulation and/or verification. Through model analysis—the counterpart of binding-time analysis in traditional use of partial evaluation for general purpose software, it can discover the execution context for the model and the components (called actors in Ptolemy terminology) contained within. It then generates the target code specific to the execution context while preserving the semantics of the original model.

3.3 Message Passing Interface

MPI is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes [8]. It's portable and programming language independent. MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers of the reference model, with socket and TCP being used in the transport layer.

We use a C library provided when including *mpi.h* in our code. A set of library calls is provided to us to use to communicate with different processors. Each processor is assigned a unique processor id, and can be used in the program to execute unique instructions. We use the MPI library to generate code targeting architectures with distributed memory and no shared memory.

4 Code Generation Framework

We overlay our MPI code generation framework on top of the Ptolemy II code generation facility [7]. The framework consists of three main stages: *partitioning*, *code generation*, and *tuning*. First, we assume that we are given a process network model where each actor and connection is annotated with a node and edge weights, respectively. These annotations should correspond to the amount of computation and communication overhead each block and connection represent. We can use these weights to influence decisions made in the partitioning stage. They also serve as parameters for us to later explore the partitioning design space.

In the partitioning phase, we analyze the model and generate clustering information using the weights and the given number of processors. It replaces weights information with processor ID's (rank) and MPI communication buffer ID's. The code generation phase infers the necessary information from these partitioning and buffer annotations. It then generates an MPI program instance which is one particular implementation of the model based upon the given partitioning. The tuning phase takes advantage of the fact that we can change the partitioning to create multiple program instances. It first executes the generated program and receives profiling information from each processors. These profiling statistics is the feedback we use to further adjust the edge and node weights. The tuning gives the partitioner a better estimate of its parameters, thus a better partition. Our framework aims to optimize the generated code by iterating over the these three phases until a fixed-point is reached.

We see our framework as the bridge between the logical design space and implementation space. Our framework applies the platform-based design principle [15] at the software design level. It allows us to adjust distributed communication parameters such as buffer sizes and the mapping of logical blocks to physical processing units. Its feedback mechanism provides the tool to systematically explore the implementation space.

5 Implementation

5.1 Partitioning & annotations

As indicated in the previous section, partitioning is the first step towards generating code that runs on parallel architecture.

As mentioned earlier, the framework where we built our code generator is Ptolemy II, which implements actor-oriented programming approach. Take the following program as example, the functionality of the program is expressed in terms of actors and connections.

The most intuitive idea of an actor is simply a block of code that performs some kind of operation. This operation may or may not depend on inputs, and the actor can subsequently produce outputs. Connections between inputs and outputs of actors indicate that data is transferred from one actor to another. One main point of these actors is that they have states associated with them. This makes dynamic balancing more difficult, because if we wish to allow processors to fire actors however we wish, then we need to ensure state transfers between these actors are done to preserve the correct actor firing behaviors. This leads to the requirement of implementations such as mutex locks to ensure sequentially consistency for these state transfer are done correctly. Due to these complications, we decided to forgo dynamic balancing first, but instead, statically partition the actor model such that each processor will only be responsible in processing specific subset of the actors, and only data transfers between the actors are needed. This data transfer could be done through shared memory or message passing, but the exact detail and choice of which is discussed already.

The problem of how to optimally map the actor model to a set of processors is a graph partitioning problem. In this problem, graphs are represented by nodes and edges. Each of these nodes and edges could have a label associated with them. In other words, nodes and edges are weighted. The goal of the algorithm is to find a partition of the nodes such that the standard deviation of the node weights among all partition is minimized, and the edge weights across the partitions is also minimized. This means if each actor in our model is a node, each communication is an edge, and we let node weights correspond to the amount of computation associated with a particular node, the edge weights correspond to the amount of communication between the actors, and finally each partition would be the set of actors mapped to a particular computation node; Then a graph partitioning algorithm would provide us with a way to maximize load balancing (i.e., minimizing the standard deviation between the sum of node weights across all processing platforms), and minimizing communication across different computation nodes.

Since the graph partitioning problem is a classic problem [12] that has been very extensively studied, we went online and found a set of software that implements the algorithm. Examples of these software are: Chaco [9] [10], SCOTCH [13], etc. Among them, we chose to use Chaco graph partitioner.

One of the main reasons we chose to use it is that compared to other graph partitioning packages, Chaco is completely open source. This is of contrast to some of the other software packages, which require the user to first sign agreements, or only allows use for educational purposes. Since Ptolemy II is a free open source software, any software we use as plug-in would ideally of the same or similar license. Another reason is because Chaco's input and output file formats are very simple, while the input could be easily generated through an actor model, the output could also have different formats that could be parsed easily. Though Chaco is chosen, any graph partitioner could theoretically be integrated with Ptolemy II in order to perform graph partitioning.

One major challenge for us is that exact node weights and edge weights are difficult to obtain. To summarize, each actor's weight should be: computation needed to fire an actor once x number of times this actor is fired. An edge weight should be: the communication needed to transmit data once across the network x the number of times the communicate linked is used. Now the computation needed to fire an actor once and the communication needed to transmit data once can be easily determined. However, the number of times an actor is fired or the number of times the communication link is used is also dependent on the behaviors of actors upstream from it.

We currently envision four solutions to this problem. In first solution, we statically traverse the graph and make use of actor informations to determine the node and edge weights. Our second solution involve simulation of the actual system on some other platform in order to determine the possible node and edge weights prior to code generation. This solution could be easily supported by the Ptolemy II framework we are currently building on. For our third solution, we could run our model for some fixed iterations, and profile the run times for each actor, and also the number of times a particular communication link is used. This information could then be used to find the node weights and edge weights for the model. Our final solution involves use dynamic balancing to solve our problem. In which case the edge and node weights doesn't matter as much.

Our decision is to use the third solution. The disadvantage of this scheme is that we need to run the model for at least one iteration before knowing what the node and edge weights should be. So in the case if the model is very complicated that doing one iteration takes very long, then the cost to come up with these values are high. However, this scheme also has a major advantage over the first and second scheme in the sense dynamic balancing is not needed, and also, since the model is run in the real platform, this provides us with platform-dependent information on the node and edge weights, which is very difficult to obtain through the other solutions.

Currently, the user needs to make these node and edge weight annotations on the actor model themselves, but we envision at a later stage, especially when we support other ways for the user to determine node weights and edge weights, we could automatically integrate a node and edge weight generation tool. We will also discuss this in more detail in the Feedback and tunning subsection later.

Given a model from Ptolemy II, we traverse the model to generate a file that looks like this:

```
NumVert NumEdge {1}[1](1)
{VertID} [vertWt] neighbor1 (EdgeWt1) ...
...
...
```

Given this input file, Chaco could be run to generate a graph partition. The Chaco software asks a number of different parameters that the user could tune in order to obtain the best partitioning. We do not discuss these options here, other than noting that the number of vertices to coarse down to is the number of processing elements available. The output will return the partition, where the output is of n lines, where n is the number of actors in the original system. Each line contains a number, which indicate the processing element this actor belongs to. After the output of Chaco is created (this output file is designated with the suffix ".out"), a *transform* function is performed, where the ".out" file is parsed and proper annotations are made on the actor model in Ptolemy for further use in later implementation. Specifically, the following annotations are made. Each actor is annotated with the partition it belongs to. We use this information to indicate which edges would represent inter-processing element communication. This information is annotated on the ports of the actors, where it could easily be determined which ports are senders and which ones are receivers. Finally, as we will discuss in the next section, we need to annotate the inter-processing element ports with a distinct number for the use of tag matching, in our MPI implementation. Thus as each inter-processing receiver port is identified, the receiver is annotated with a unique identification integer. The reason why we only annotate this information on the receiver will be discussed in the next section.

5.2 MPI code generation

After analyzing and annotating the actors and ports accordingly, we now start generating MPI source code from the model. The general structure of our generated code is shown in figure [1]. Each actor has it's own firing function which is called in the function *main*. The partitioning is done by a set of *if* statements. The variable *rank* is the unique number assigned to each processor that is used to identify itself. It is also how the processors know which actor to execute. The *while* loop will invoke each actor function continuously, and inside the actor function is where we determine whether an actor can fire or not.

The structure of the actor functions is shown in figure [2]. Each actor has four parts. First, it tries to receive data. Second, it checks whether it is safe to fire the actor. Most cases, this means whenever an input data has arrived. Third, it fires the actor by running the actor function code. Finally,

```
int main(int argc, char *argv[]) {

  initialize();

  //MPI initialization code
  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size (comm, &n_proc);
  MPI_Comm_rank (comm, &rank);

  //Main loop that loops through the actors
  while(1) {
    if (rank == 1) {
      Actor1();
    }
    if (rank == 2) {
      Actor2();
    }
    if (rank == 3) {
      Actor3()
        }
    if (rank == 4) {
      Actor4();
    }
  }
}
```

Figure 1. Main Body Code

the actor sends out the data produced to its connecting actor. Below we describe the details of communication from Process Network to MPI.

5.2.1 Buffer Allocation

For each communication channel, Process Networks imply an infinite sized FIFO queue. Of course in reality there is no such thing as infinite memory space, so we need to allocate buffers for each communication channel, and block the actors when theses buffers are full. In a shared memory system, all processors have access to the same memory, thus we could easily allocate a buffer for each communication channel, and each actor would be able to access them. However, since we are targeting a message passing system, the processors have no shared memory between them, so every communication across processors is handled by MPI send and receive requests.

Since it's possible to have more actors than processors, more than one actor could be mapped to a single core. If the actors on the same core communicate with each other, then we simply allocate a circular buffer for the communication channel globally, so both actor functions can access this buffer. Along with the buffer, we create a buffer header

```
void* ActorFunction() {

  //Code that checks if any input tokens
  //are available
  ReceiveInput();

  //Check if the actor can fire based on
  //the semantics of Kahn Process Networks
  if ( HasInput() && CanOutput() ) {

    //Actor firing code
    FireActor();

    //Send out tokens to the connecting actor
    OutputToken();
  }
  return;

}
```

Figure 2. Actor Code

which allows the reader and writer to read from and write to the buffer. Since we are not running multiple threads for actors on the processors, we don't need to worry about race conditions.

When actors communicate across processors, we could either allocate the buffer on the sending side or the receiving side. We chose to allocate it on the sending actor because it would cost an extra message if the receiving actor had to somehow notify the sender that the communication buffer is full. On the sending side, the actor can detect whether the send buffer is full. If it's send, then the actor can't fire, and needs to wait for the send requests to be cleared in order to fire and send more tokens. Note this send buffer is separate than the MPI send buffer. We have no control over the buffer space which MPI allocates and uses for its transmission protocol. But after we call an MPI send using data from the buffer, we cannot overwrite the contents until we are sure MPI sent out the data. This is why we need an additional buffer, so the actor can continuously fire and put data onto the send buffer, while MPI send calls take data off the send buffer and move them to the MPI send buffer. More details about the communication calls follow below.

5.2.2 Communication Channel

In this section we will go into the details the calls relating to communication. As identified before, there are four distinct types of communication. Send and receive from the local processor, and send and receive from separate processor.

Before an actor can fire, if the actor fire function produces a token that needs to be sent out, it needs to check if the send buffer is full. For local communication channels, we check the header of the local buffer. The header contains two indexes for the buffer, a read index and a write index. Whenever data is put onto the buffer by the sender, the write index is incremented. If data is read by the receiver, then the read index is incremented. The indexes loop around to create a circular buffer. The check to see if the buffer is full is simply checking if the write index is right behind the read index. This indicates that there are no more slots to write because the next slot hasn't been read yet. Because it's a circular buffer, we also make sure that if the read index is at slot zero, the buffer is full when the write index is at the last slot. For the receiver, checking if the buffer has data to read is just making sure the two indexes aren't on the same slot. Everything would indicate that there is more data to read. We created two functions *hasLocalInput(BufferHeader)* and *isLocalBufferFull(BufferHeader)* to complete those checks.

For actors that receive data from another processor, we use an asynchronous receive MPI call to receive the data. Even though the semantics of Process Networks state that actors do blocking receive, but we cannot use a blocking MPI call because we are only running one thread on each processor, so other actors need a chance to run their actor function. If we were to use a blocking receive, then the executable would easily deadlock because one actor waiting for its input token would completely halt the other actors on the same processor from firing. So to create the illusion of the blocking effect without actually calling a MPI blocking receive, we call MPI test after the MPI asynchronous receive. We use a flag to indicate whether the asynchronous call has been made. Then, we use the results of MPI test, which returns a logical integer to indicate the status of the receive, to determine whether the MPI asynchronous receive has received data. Once the MPI test call has returned a true, then we can safely fire the actor. Because the communication buffer is allocated on the send side, we don't need to allocate extra buffer on the receive side, just one variable to actually receive the data from the MPI receive call.

For sending over MPI, we fully utilize the send buffer allocated on the send side. During the firing of an actor, we use a MPI asynchronous send to send out a data from the send buffer over MPI. Because we cannot modify the buffer contents until we are sure MPI has sent it out, we also need to use MPI test statements to insure that the requesting send has completed, before we write new data onto the send buffer. However, to lower the overhead and take advantage of the send buffer allocation, we only need to test when the send buffer is full. Conceptually, what we do is to continuously send out tokens as soon as we produce them using MPI asynchronous send. The send buffer will keep track of the amount of send requests we have sent. Once the buffer is full, we use a MPI_Testall call to test the whole

send buffer request. MPI_Testall will return the indexes in which requests have finished, and we reuse those slots for new send requests. If no slots are cleared, then we don't allow the actor to fire.

Actors may also have more than one input port and more than one output port, which will map to different communication channels. So most of the time a combination of the above communication schemes are used. Figure [3] shows the actual code for the MPI communications.

```
/****Code block for MPI Send and Receive*****/
void* ActorMPI() {
  //Receive variables
  static int MPI_recv = false, Recevied_Input = false, recvTag = 0;
  static MPI_Request Receive_request;
  static double Receive_Data;

  //Send variables
  static struct mpiBufferHeader MPIBufferOutHeader;
  static MPI_Request Send_requests[MPI_SENDBUFF_SIZE];
  static double MPI_sendBuffer[MPI_SENDBUFF_SIZE];
  static int sendTag = 1;

  //Receive code
  if (!Recevied_Input) {
    if (!MPI_recv) {
      MPI_Irecv(&Receive_Data, 1, MPI_DOUBLE, src, recvTag, comm,
&Receive_request);
      recvTag += NUM_MPI_CONNECTIONS;
      recvTag &= 32767; // 2^15 - 1 which is the max tag value.
      MPI_recv = true;
    }
    MPI_Test(&Receive_request, &Received_Input, MPI_STATUS_IGNORE);
  }

  //Clear send buffer when its full
  if (isMpiBufferFull(mpiBufferHeaderOut)) {
    //MPI Library call
    MPI_Testsome(MPI_SENDBUFF_SIZE, Send_requests...);
    MPIBufferOutHeader.current = 0;
  }

  //Check to see if it can fire
  if (Received_Input && !isMpiBufferFull(mpiBufferHeaderOut) ) {
    //Fire code
    output = Fire(Receive_Data);

    //Send through MPI
    MPI_sendBuffer[MPIBufferOutHeader.current] = output;
    MPI_Isend(MPI_sendBuffer[MPIBufferOutHeader.current], 1,
              MPI_DOUBLE, dest, sendTag, comm,
&Send_requests[MPIBufferOutHeader.current]);
    MPIBufferOutHeader.current++;
    sendTag += NUM_MPI_CONNECTIONS;
    sendTag &= 32767; // 2^15 - 1 which is the max tag value.
  }
}
```

Figure 3. MPI Communication Code

5.2.3 Global Tag Scheme

To insure that the receiver receives the correct data from the sender, MPI uses a tag scheme where matching tags are used to determine which data is being received. However, without shared memory, both sender and receiver need a way to know which tag to send and which tag to receive separately. Also, when multiple actors on multiple cores are cross communicating, the tasks of keep tag could be daunting, especially when multiple MPI calls are used from multiple processors. We used a simple scheme to solve this problems.

As mentioned above, each MPI connection is annotated with a unique id. This id will help us in our tag scheme. We recognize the total number of MPI connections, and use that as a global increment number. The unique id is generated by a count upwards from 0, so the largest unique id will be the total number of MPI connections minus one. For both MPI send and MPI receive calls, the tag number will start with the unique id. After every send or receive, we increment the tag by the total number of MPI connections. This allows no overlapping of the tags, and as long as we make sure the corresponding MPI send and receive calls start on the same tag (unique id), they will both use the same set of tag numbers (tags loop around after 2^{15}) when they increment.

5.3 Feedback and tuning

As we mentioned in the previous sections, we implemented a scheme to profile the generated code in order to come up with the node and edge weights of a graph. The idea behind this implementation is simple, where an arbitrary set of node and edge weights could first be chosen, and annotated on the graph. Code generation is then performed on the model. The generated code could then be run for one complete iteration. The amount of time used to perform computation on each actor within one iteration and the number of times each communication link is utilized could be profiled, and translate to node and edge weights. One point we need to make is that only the time to perform the actual computation of each actor is profiled, but not the amount of time to perform the overhead in inter-platform communication and in figuring out whether the actor could actually fire.

This is because we do not yet know of a way to properly characterized this overhead. For example, an actor that will only fire once may have a very large overhead compared to its actual computation since it may check many times whether it can fire only to find that it cannot. Since this overhead is not part of the actual computation, we do not include it as a part of the profiling. This means that even if we assume the final partition is the most optimal one in minimizing communication and maximizing load balancing (Graph partitioning is a NP-hard problem, and Chaco does not guarantee the output will be the most optimal partition), it could be that a better partition exists taking into account of the overhead we just discussed. We note that this is in itself a research problem, and since it's not the focus of this project, we only have primitive support for solving it, where we let the user manually annotate the model with nodes and edges weights, and let them have the freedom to play with the values in their liking.

Another complication of this scheme is that it may not

work all the time for process network models. This is due to the fact Process Network model of computation, unlike more deterministic ones such as Synchronous Dataflow, does not guarantee that one could always find a firing pattern that would define one "iteration" of firing. This is ongoing research and we are still looking for ways to properly solve this problem. However, we do note that this is not an artifact of the solution we chose to approach the problem finding node and edge weights. As we mentioned in the previous section where a total of four solutions were given to solve the problem of finding proper node and edge weights, only the last one, dynamic load balancing can escape this problem of not having to define "iterations". Since we do not yet support dynamic load balancing, it is a problem we simply have to live with at this point of the project, and we wish to introduce dynamic load balancing at a later point of the project. We also note that the examples we used in this paper to simulate the results all have the property of allowing the user to properly indicate what they mean in one "iteration" of model firing.

6 Results

To show the results of our code generator, we picked two applications, and ran them on a computing cluster. We will discuss the details of how we obtain results in this section.

6.1 Application

The application we chose to test our code generator for parallel platforms was chosen mainly because they are very mathematically focused, which is a common property among all parallel applications. This application generates a butterfly graph on transcendental plane. The butterfly model is a famous transcendental plane curve discovered by Temple H. Fay [5]. The polar equation of this graph is given by:

$$r = e^{\sin\theta} - 2\cos(4\theta) + \sin^5\frac{2\theta - \pi}{24}$$

In Ptolemy II, this graph could be computed by the following model.

We could see that there is intrinsic concurrency in that four "paths" lead to the add-subtract actor, where computation could be done in parallel.

6.2 Platform

The platform we chosen to run our generated MPI code on is the Jacquard cluster in National Energy Research Scientific Computing Center, at Lawrence Berkeley National Laboratory. Jacquard is an Opteron cluster that runs a Linux operating system. Statistics about this cluster could be found at [2].

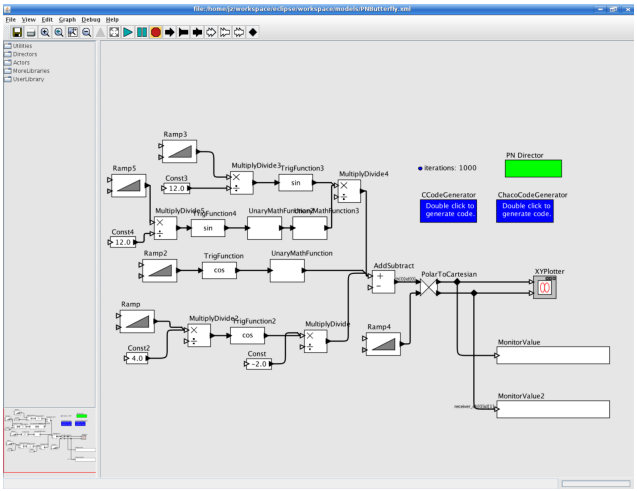

Figure 4. Butterfly Curve Model in Ptolemy

However, We would like make a note of the following interesting points about this cluster. First, the number of processor per node is two. This means that for the Pthread implementation, two is the only number of processors we can test on. Another interesting data are the MPI latency and bandwidth, which we will use later on to analyze our data.

One note about this platform is that we were not able to implement a plotter on the cluster. Thus the *XYPlotter* could not generate proper code to run Jacquard, and this actor had to be deleted from the model. However, since *MonitorValue* actors are implemented with simple "printf" statements, the results from the generated code could still be compared to the simulation in order to ensure the generated code was behaving correctly.

6.3 Results and Analysis

Through the workflow as detailed in the previous section, we have generated code for the Butterfly model and confirmed with simulation that the values obtained were coherent with the simulated values. The table below are the simulation times from our result.

For better visualization, this data is also plotted into the figure 5.

This data shows the comparison between the generated code that uses the Pthread library for shared memory architecture, and the one that uses the MPI application programming interface for message passing. As we have said, only two nodes of shared memory is available, thus data from the Pthread version of the code is only benchmarked with two cores. For the MPI version of the code, we only tested for three partition numbers of 2, 3, and 4. This is mainly due to a flaw in choosing our application, as we will elaborate

Table 1. Jacquard Cluster Characteristics

Number of cores	MPI 500 Iter	MPI 1000 Iter	MPI 2500 Iter	MPI 5000 Iter	Pthread 500 Iter	Pthread 1000 Iter	Pthread 2500 Iter	Pthread 5000 Iter
2	23.0 (ms)	49.0	137.6	304.0	17.9	47.1	182.0	406.0
3	18.8	37.4	95.4	195.0				
4	19.4	38.3	97.5	193.0				

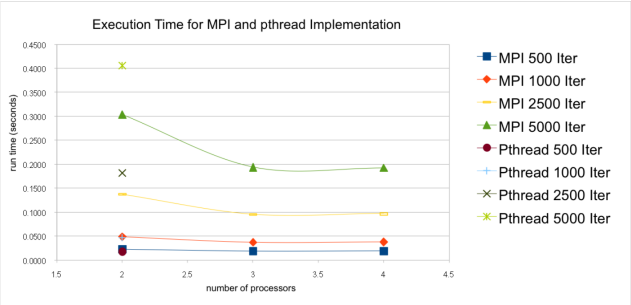

Figure 5. Results on Generated Butterfly Curve Model

later.

In benchmarking of this application, upon the first run of the code with a random partition, As we mentioned in the previous section, for this model, we could easily define the number of *iterations* each actor could be fired. In fact, in this model, each actor is fired exactly once in each iteration. According to the Process Network semantics, actors such as *AddSubtract* would only fire if both of its input ports are filled, and upon firing, it will consume all data tokens available at its inputs. By running the generated code in MPI, we found that the monitor values are really the bottlenecks in the performance of this implementation. This actually should not be surprising since the "printf" statements are basically I/O calls that could take a considerably amount of time to executing. To make their effects less obvious, we output to a file instead of the screen, however the execution time of *MonitorValue* is still around 100 times that of the *Const* actor. We gave the proper node and edge weights, and as long as we have at least three partitions, two partitions would be mainly responsible for executing the *MonitorValue* actors, and the other actors are equally partitioned to the rest of the processors. A set of results were obtained.

We ran our model for different number of iterations, ranging from 1000 to 5000. Unfortunately, also because the *MonitorValue* actors are the bottlenecks, no difference could be observed comparing the MPI and the Pthread implementation, since both of them would have one processor running one of the "printf" statements almost all the time. Because of this, once we have more than two partitions, we do not see any increase in the execution time.

Also, comparing the results between MPI and Pthread implementation, we see that for small number Iterations, Pthread out performs MPI, but it becomes the other way around for a large of number iterations. One explanation for this phenomenon is that since the pthreads are scheduled by the OS, as the number of iterations increase, the amount of overlap between runs between the two iterations also increase. While in MPI the allocation is done statically, and actors fire whenever there are events to process, the OS is probably trying to do dynamic balancing in order to make sure when an actor is blocked, it fire actors in other threads. This incurs an overhead in the Pthread implementations. However, since the two *MonitorValue*'s are the main overhead, which are mapped to each processor in the MPI implementation, the overhead spent in dynamically balancing the threads in Pthread implementation is probably outweighs the gain in dynamically balancing.

Interestingly, in the Pthread implementation, as the number of iteration scales, the increase in computation time is not exactly linear. However we suspect this is purely an artifact of the operating system that schedules this thread. We still expect the main trend in the computation time vs. number of iterations to scale mainly linearly in both the Pthread and MPI implementations. In the MPI implementation, the only change that could result in the increase of computation time not being linear are the MPI buffers. Since Process Network models are indeterministic in when exactly each actor checks its input port to see if data tokens are available for processing, it could be that some actors are fired more often than others during some period of time. This may mean that actors might be starved a bit during the execution, and thus lead to indeterministic results in the execution time of the model. In the Pthread implementation,

In retrospect, we should have picked a "better" application to test our code generator. By "better", we mean an application that has more coarse level concurrency. This is because the advantage of firing these actors of small computation intensity in parallel is offset by the overhead in message communication. Even though we have gained some interesting insights in the performance and overhead between MPI and Pthread, we probably could have gotten other insights by choosing an application that does not include two actors that are I/O operations that takes all the computation

power. Instead, if an application of more coarse grain parallelism is implemented, we could probably see better improvement for application run times, since the overhead for message passing would not be less influential.

7 Future Work

Currently for our code generation framework, we have implemented the MPI code generation engine, as presented in this paper. We plan to continue improving the code generated by this engine, as well as add new engines to target different architecture platforms. This will include code using either OpenMP library, or even UPC library, to target shared memory systems. Along with that, we want to be able to generate code from different high level languages, not just Process Networks. The Ptolemy project studies the interaction between heterogeneous semantics of high level specifications called "Models of Computations." We plan to take full advantage of this infrastructure and be able to generate code from different models of computations, and even a mixture of them. This will give the users more expressiveness to express their application and design. We also want to study and research more on the feedback we can provide to the users used for tuning and optimization of performance.

8 Conclusion

In this paper we present a framework used to quickly explore and design applications on parallel platforms. We do so by allowing users to express their designs in higher level languages, and then use code generation to target parallel platforms. The framework we presented allows for flexibility in design, and also the ability to quickly explore and compare various design parameters, including number of cores, partitioning of applications onto cores, and memory system etc.

We showed our implementation of the MPI code generation engine from Process Network, and ran the generated code on the NERSC computing clusters to see the resulting performance and overhead. When comparing with the current pthreads implementation for the applications we tested on, it showed lower overhead for larger iterations. In our butterfly application, scalability was thwarted by the I/O operations, but we believe that for more computation intensive applications, scalability can be achieved.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] N. E. R. S. C. Center. Jacquard opteron cluster. website, Oct 2007.

[3] J. Davis. Ptolemy ii - heterogeneous concurrent modeling and design in java, 2000.

[4] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[5] T. H. Fay. The butterfly curve. *American Mathematical Monthly*, 96(5), May 1989.

[6] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[7] M.-K. L. Gang Zhou and E. A. Lee. A code generation framework for actor-oriented models with partial evaluation. In Y.-H. L. et al., editor, *Proceedings of International Conference on Embedded Software and Systems 2007, LNCS 4523*, pages 786–799, May 2007.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[9] B. Hendrickson and R. Leland. The chaco user's guide — version, 1994.

[10] B. Hendrickson and R. W. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing*, 1995.

[11] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[12] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[13] F. Pellegrini. Scotch 3.1 user's guide.

[14] J. L. Pino, T. M. Parks, and E. A. Lee. AUTOMATIC CODE GENERATION FOR HETEROGENEOUS MULTIPROCESSORS. In *Proc. ICASSP '94*, pages II–445–II–448, Adelaide, Austrailia, 1994.

[15] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, 2002.

[16] G. C. Sih and E. A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):625–637, 1993.

[17] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *International Conference on Compiler Construction*, 4, 2002.

[18] P. Warner. Network of workstations active messages target for ptolemy c code generation. Technical Report UCB/ERL M97/8, EECS Department, University of California, Berkeley, 1997.