A Prototype Embedded Microprocessor Interconnect for Distributed and Parallel Computing

Bryan Hughes
Electrical and Computer Engineering, Texas Tech University
Lubbock, TX 79409, United States

and

Brian Nutter
Electrical and Computer Engineering, Texas Tech University
Lubbock, TX 79409, United States

and

Per Andersen
Computer Science, Texas Tech University
Lubbock, TX 79409, United States

and

Daniel Cooke
Computer Science, Texas Tech University
Lubbock, TX 79409, United States

ABSTRACT

Parallel computing is currently undergoing a transition from a niche use to widespread acceptance due to new, computationally intensive applications and multi-core processors. While parallel processing is an invaluable tool for increasing performance, more time and expertise are required to develop a parallel system than are required for sequential systems. This paper discusses a toolkit currently in development that will simplify both the hardware and software development of embedded distributed and parallel systems. The hardware interconnection mechanism uses the Serial Peripheral Interface as a physical medium and provides routing and management services for the system. The topics in this paper are primarily limited to the interconnection aspect of the toolkit.

Keywords: Parallel Computing, Interconnection Networks, Embedded Systems.

1. INTRODUCTION

Although parallel computing has been around for decades, it has only been used in niche high performance computing (HPC) applications until recently. With the advent of multi-core processors and new computationally intensive applications, such as High-Definition (HD) video processing, parallel computing is becoming mainstream. Unfortunately the development tools available to implement these applications, especially in the embedded market, have remained relatively unchanged since the mid 1990s.

To implement an embedded parallel computing system using current technology, one would first have to implement an interconnection mechanism. Current pre-made, high-end options include HyperTransport™ and RapidIO®, while current low-end options include the Inter-Integrated Circuit (I^2C) bus and the Controller Area Network (CAN) bus. The high-end solutions work well for parallel computing with bandwidth rates up to 20.8 GB/s on HyperTransport[1] and 10 GB/s on RapidIO[2]. However, these interconnects are expensive to implement because very few microcontrollers contain the necessary interface circuitry internally, which means they would require additional external circuitry. The bandwidth offered by these interconnects is also significantly higher than most microcontrollers can process. The low-end solutions are too antiquated to support the communication demands of parallel computing, with bandwidths of 3.4 Mbps for I^2C[3] and 1 Mbps for CANbus[4]. For node-to-node bandwidth, this might be an acceptable rate, but in I^2C and CANbus, these bandwidths are for the entire system, which gives a node-to-node bandwidth equal to the total bandwidth divided by the number of nodes during heavy load. We require a new type of interconnect that most microcontrollers can support without external interface IC's that still provides adequate performance for parallel computing.

The toolkit proposed here utilizes the Serial Peripheral Interface (SPI) protocol as a physical layer. A protocol has been developed to sit on top of SPI that provides routing, guaranteed delivery, and other services for up to 256 nodes. A prototype router is being developed for the protocol using a TMS320F2808 DSP Controller from Texas Instruments (TI) with 4 communication links that can operate at up to 25 Mbps. A subset of the Message Passing Interface (MPI) will be developed to take advantage of the protocol. This sub API will

serve as a middleware provider for SequenceL, a functional programming language being developed at Texas Tech University that features automatic concurrency. Once the toolkit is completed, the time necessary to develop a complete embedded parallel system should be greatly reduced.

2. INTERCONNECTION HARDWARE

To create an interconnection for embedded systems, a different perspective from computer interconnection is required. Adding new interconnection functionality to a PC is as simple as spending 10 minutes installing an expansion card. However, adding new interconnection functionality to an embedded system must be done at design time. This process requires designing printed circuit boards (PCB's) and writing software that allows microcontrollers to take advantage of the hardware, which is a non-trivial task. The ideal interconnection for embedded systems would take advantage of hardware that is common to most microcontrollers on the market so that designers can skip the entire interfacing step. At the same time, the interconnection must be reasonably fast in comparison to the clock speed of the microcontroller. While 'reasonably fast' is a relative term, picking a base communication link speed that is within an order of magnitude of the microcontroller's primary clock speed is probably a good choice. SPI was chosen because it is found on most microcontrollers on the market, and can typically run at clock speeds up to 1/4 or 1/2 of the clock speed of the microcontroller.

Physical Layer Signaling
The SPI protocol is a master-slave, point-to-point, full duplex, serial protocol. It consists of three or four signal lines: transmit, receive, clock, and an optional slave select. The slave select signal is not used because its behavior tends to vary from implementation to implementation. A master-slave protocol is not really ideal in parallel computing because nodes in a parallel system should be equal peers. To get around the master-slave limitation, nodes will all be slaves by default, and will dynamically elevate themselves to a master whenever they have information to transmit. This mechanism loses the full-duplex capabilities of SPI, but allows all nodes to be true peers. The four signal lines for this peer-to-peer SPI are defined in Figure 1. The Slave-In/Master-Out and clock signals behave the same as their standard SPI counterparts. An arbitration scheme

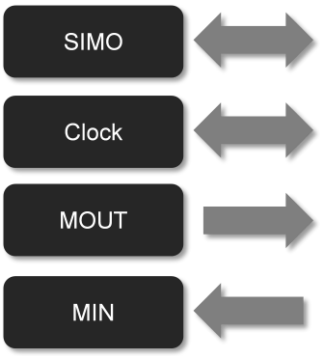


Figure 1: Link Signal Definitions

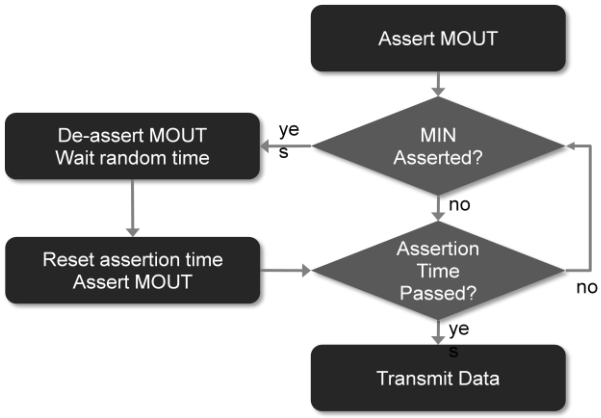


Figure 2: Elevation Process

has been created to prevent nodes from talking over each other that uses the masters in and out signals, MIN and MOUT respectively, according to the state diagram shown in Figure 2. This mechanism is modeled after Ethernet's back-off-and-wait-randomly collision mechanism.

Routing Boards
To aid in the development of the toolkit software, individual routing boards are being designed that allow rapid prototyping of network topologies. These boards also serve as a reference design for the routing chips. Each routing board consists of a single routing chip, support circuitry, and headers for connecting with other routing boards. The block diagram for the board is shown in Figure 3.
Each board contains its own power regulation circuitry centered around a TI TPS70102 dual voltage linear regulator, four link headers for connecting to other routing boards or hosts, a set of switches for configuring the board, a seven segment display for debugging purposes, and a Joint Test Action Group (JTAG) port.

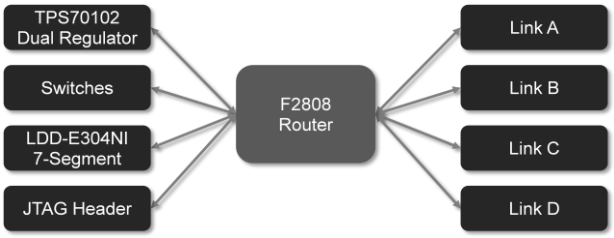


Figure 3: Routing Board Block Diagram

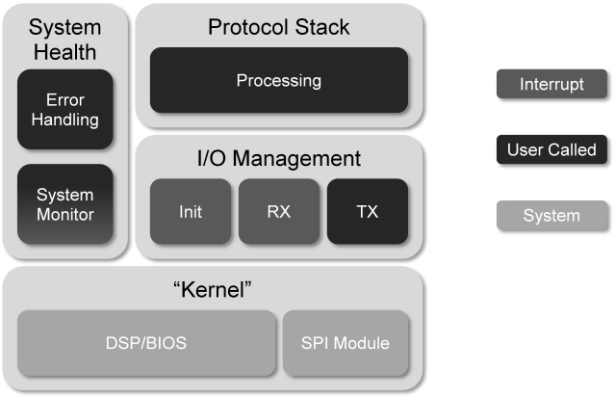


Figure 4: Software Modules

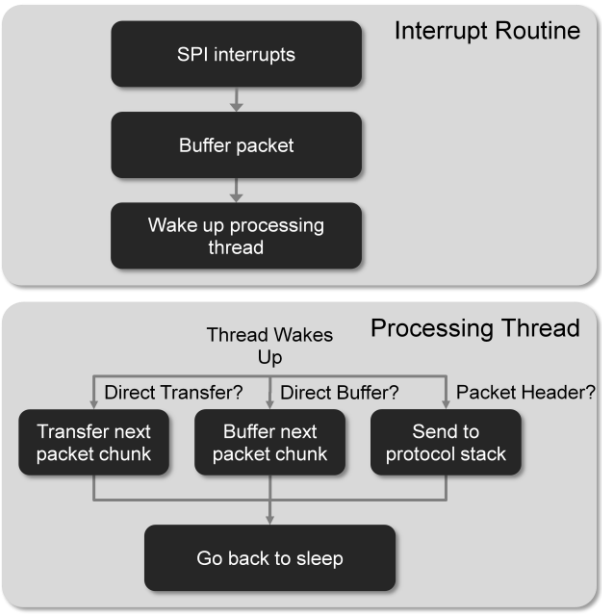


Figure 5: Receive Flow Chart

Router Software Architecture

The routing software is divided into four modules: the "kernel," I/O management, Protocol Stack, and System Health. The relationships of these modules are shown in Figure 4. The system health module controls error handling and high-level management of the router. The Protocol Stack process packets and performs the appropriate action. The I/O management module serves as a driver for the peer-to-peer SPI port. The "kernel" consists of the Real Time Operating System (RTOS) DSP/BIOS by TI and the SPI subsystem. Note that DSP/BIOS does not interface with any SPI ports. The primary reason for using DSP/BIOS, despite its lack of SPI drivers for this particular chip, is for its threading capabilities. Each port has a thread dedicated to processing incoming packets. This allows the routers to handle multiple streams simultaneously. These threads exist to offload most of the computation from the SPI interrupt routines. The receiving process is shown in Figure 5. The interrupt itself is designed to be as short as possible because interrupts block execution in the rest of the system. When a 128-bit chunk of a packet comes in and the interrupt occurs, the interrupt routine buffers the packet and wakes up the processing thread. If the buffer is full, the interrupt routine will assert the master out pin for that port to prevent the other node from sending any more information. Once the interrupt has finished and normal threading has resumed, the processing thread then performs the actual processing of the packet chunk. If the packet chunk is the header of a new packet, it is sent to the protocol stack for processing. If the packet chunk is not a header, then it is part of a continuing transfer and is routed accordingly. After the processing thread is done processing the entire packet buffer, it goes back to sleep. Because each port has its own processing thread, multiple packets on multiple ports can be processed simultaneously.

Handling Data Transfers

If a packet doesn't have a payload, the system sends the data chunk to the protocol stack for processing because the entire packet is contained in the data chunk in question. If a packet has a data payload, the system must know what to do with each 128-bit data chunk that comes through after the header. The initialization of a multi-chunk transfer is shown in Figure 6.

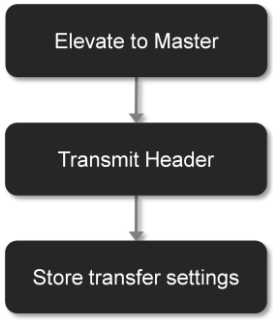


Figure 6: Multi-Chunk Transfer Initialization

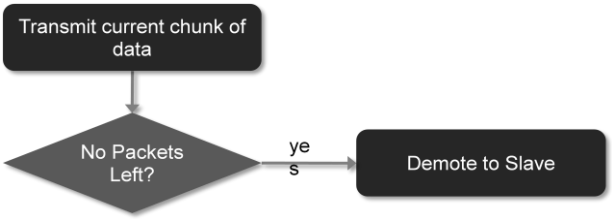


Figure 7: Multi-Chunk Transfer

After transferring the packet header using the normal transfer routines, the transfer settings are saved so that when the next chunk comes in, the chunk won't be sent to the protocol stack but can be transferred directly, based on the saved settings as seen in Figure 7. When the data payload is destined for the router in question, it behaves like a multi-chunk transfer except that it sends the data to a buffer rather than to another port.

3. COMMUNICATION PROTOCOL

A custom protocol is being designed to take advantage of the network. Because the protocol is only required to handle communication for at most 256 nodes over distances no more than a meter or so, the protocol doesn't need to be as complex as Ethernet/IP/TCP/etc. By simplifying and condensing the packet structure, the overhead goes down.

Packet Header Definition

Each packet consists of a header and an optional data payload. The header structure is defined in Figure 8, and a description of the fields is in Table 1.

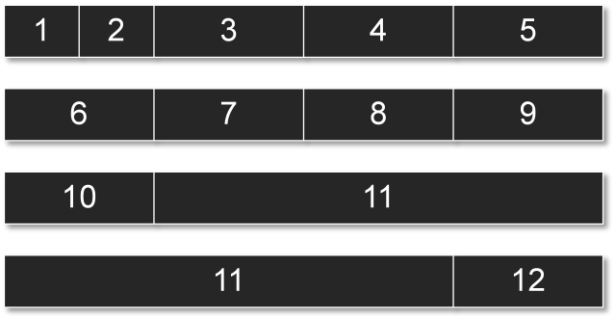


Figure 8: Packet Header Definition

Field	Name	Length
1	Version	4
2	Communication Type	4
3	Source	8
4	Destination	8
5	Command	8
6	Command Sequence Step	8
7	Number of Hops	8
8	Packet ID	8
9	Source Process ID	8
10	Destination Process ID	8
11	Command Specific	48
12	CRC	8

Table 1: Packet Field Definitions

The first field is the *Version* field, which supports multiple revisions of the protocol in play at the same time. The *Communication Type* field specifies the type of communication, which will be discussed in next section. The *Source*, and *Destination* fields define a source and destination node address.

The *Command* field specifies the packet's purpose. Example commands include transferring data, requesting a node address, getting a list of running processes on a node, etc. In Ethernet, the equivalent would be the specification of IP in the Ethernet header, and then TCP/UDP in the IP header, and then the port number specified in the TCP/UDP header, with the port number ultimately specifying the packet's purpose. While the system used in Ethernet/IP/TCP/UDP allows greater flexibility and potential for growth, it also introduces significantly more overhead as well as a more complex protocol stack, compared with using a single, non-nested protocol. Given the nature of embedded systems (little memory or processing power) a simpler, if less flexible, protocol is preferred due to its memory and computationally friendly nature. In this protocol, a command represents a sequence of steps that are to be performed for a given action. As an example, a data transfer consists of a request to send data, sending the data, reporting whether or not the data transferred without error, and retransmitting the packet if necessary. The field *Command Sequence Step* keeps track of the current step in the command sequence. Storing the sequence in the packet allows nodes to send a packet and forget about it, instead of having to keep track of all of their pending requests.

The *Number of Hops* field keeps track of how many routers the packet has passed through. This field allows a time to live limit to be set on packets. The *Packet ID* field, in conjunction with the *Source* field, allows each packet to have a unique identifier in the system. The hosts use the *Source Process ID* and *Destination Process ID* fields in the system to identify what to do with the data payload. These fields are used, because the data transfer command is a generic type; i.e. it does not specify what to do with the data. The *Command Specific* field contains data that is command specific. This field allows commands to store information directly in the header instead of in the data payload section. The benefit of this field is that the overall packet is smaller and processing the packet is a lot simpler, because the entire packet is contained in a single 128-bit packet chunk. The *CRC* field contains an 8-bit CRC of the data using the $x^8 + x^2 + x + 1$ polynomial.

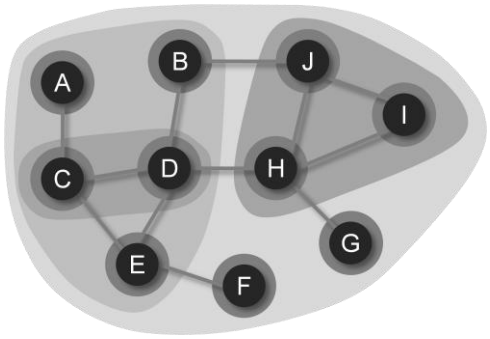
Communication Types

Six types of communication have been developed: Unicast, Multicast, Broadcast All, Broadcast Routers, Broadcast Hosts, and Addressless. In this system, routers and hosts are equals in that both have addresses and can communicate with each other directly. Unicast works as expected; it allows sending a packet from one node to another. Broadcast works as expected as well, except that a node can broadcast to everyone in the system, to all routers but none of the hosts, or to all hosts but none of the routers. Multicast in the toolkit works a little different than multicast in TCP/IP. In the toolkit, when a node wants to join a multicast group, it registers with the master router, which then notifies all of the routers. Routers only pass along multicast packets when a member of the multicast group is further along the link, as in TCP/IP. Where multicast differs from TCP/IP is that there is no multicast server. Anyone in the group can send a multicast packet to the rest of the group. In this sense, multicast works like subnets do in Ethernet, except that the multicast groups are dynamic and a node can simultaneously belong to multiple multicast groups. Addressless is used when a node powers on and needs to request an address from the master router. If a node's neighbor has an address, the node uses its neighbor as a proxy to ask the master router for an address. This method is necessary because only one address exists for nodes, and when a node powers on there is no way of identifying it. Ethernet/IP/TCP, in contrast, uses a dual addressing scheme where the IP address can be dynamic, but the MAC address is hard-coded, and so it doesn't need any form of addressless communication.

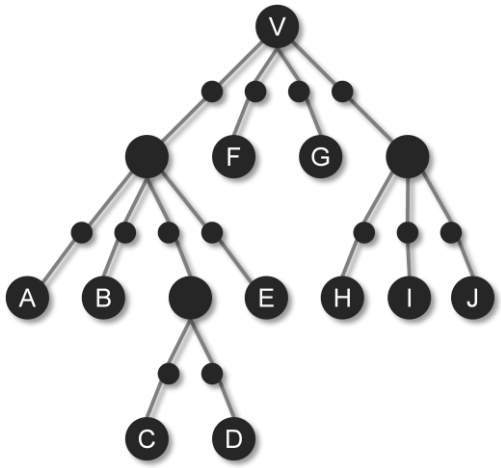
Routing Algorithm

The routing scheme used is based on Räcke's oblivious routing scheme outlined in [5]. An oblivious routing scheme is one that takes a routing request $\sigma_i = (s_i, t_i)$, with source s_i and target t_i, and produces a route f from s_i to t_i without knowledge of the global state of the network. This implies that the routing scheme is non-adaptive, i.e. it does not depend on the real-time congestion of the network. A general network, e.g. one that does not necessarily conform to a specific topology, can be modeled as an undirected graph $G = (V, E)$ with set of nodes V, set of edges E, and number of nodes $n = |V|$. Räcke's method, in short, maps G to a tree network $T_{\mathcal{H}}$, finds the shortest path on $T_{\mathcal{H}}$, and maps the result back to a set of paths on G. One of the paths in the result is selected randomly to produce route f.[5]

Tree networks are used because it is simple, even trivial, to find the shortest path in the tree network between two nodes. The tree $T_{\mathcal{H}}$ is constructed by created a root node that corresponds to V, i.e. it contains all nodes. This node is then subdivided into children nodes. These children nodes are recursively subdivided until all nodes contain a single element $v \in V$. This decomposition process forms a natural tree structure as in Figure 9(b), given a general network as show in Figure 9(a). Note that all leaf nodes contain a singleton set $\{v\} \in V$. For technical reasons, an intermediate node y_i is inserted between each natural node y_n in the tree $T_{\mathcal{H}}$. Note that in Figures 9(b), natural nodes are represented by large circles, and intermediate nodes are represented by small circles. A cluster H_v is the cluster associated with y_i and y_n.



(a) General Network G



(b) Tree Network $T_{\mathcal{H}}$

Figure 9: General to Tree Network Mapping

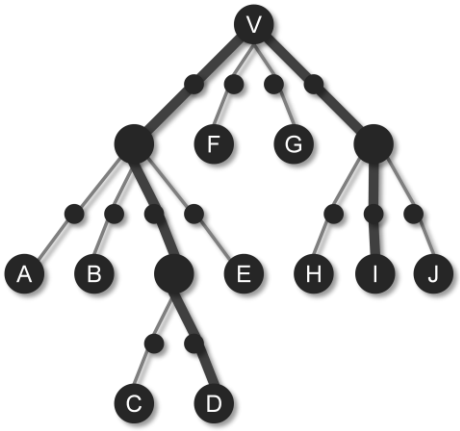
$$B_t(v_n, v_i) = out(h_{v_i}) \tag{1}$$

The bandwidth of an edge $E_t = (v_n, v_i)$, defined in Eq. (1), states that the bandwidth of E_t is defined as the bandwidth of all outgoing edges from the intermediate node in H_{v_i}. The level of a natural node in $T_{\mathcal{H}}$ is defined as the number of natural nodes on the path from v_n to V, the root node. The level of a cluster H is defined as the level of its natural node v_n. Each level is given a weight w_l.

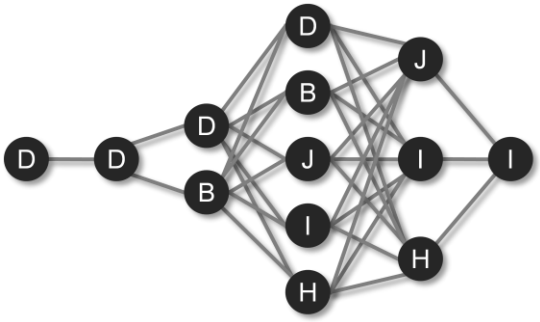
The properties of $T_{\mathcal{H}}$ discussed above are used to define a concurrent multi-commodity flow (CMCF) problem for each cluster H. A commodity $d_{u,v}$ is defined for each cluster $u, v \in H_{v_i}$, where u is the source, v is the sink and $dem(u, v)$ is the demand as defined in Eq. (2).

$$dem(u, v) = \frac{w_{l+1}(u) \cdot w_{l+1}(v)}{w_{l+1}(H_{v_i})} \tag{2}$$

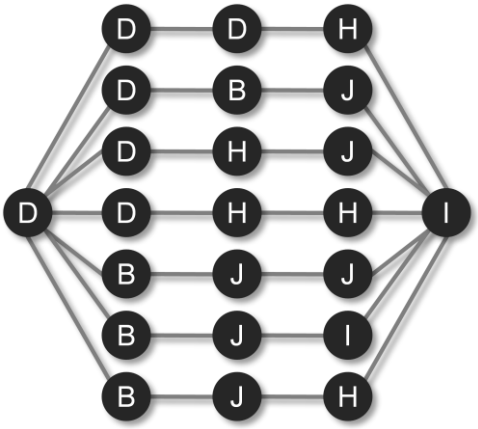
Given the CMCF for H, a path between u and v can be found by decomposing $d_{u,v}$ into a set of convex paths $P_{u,v}$. Each path $p_j \in P_{u,v}$ is assigned a weight, and one of the paths in $P_{u,v}$ is chosen based on the weight and a randomized input. An example path in $T_{\mathcal{H}}$ is show in Figure 10(a), with the associated paths in G in Figure 10(b) and the valid paths in G after solving the CMCF problem in Figure 10(c).[5]



(a) Unique $P_{u,v}$ in the Tree Network



(b) All Paths $P_{u,v}$ in the Tree Network



(c) CMCF Solution $P_{u,v}$ in the General Network

Figure 10: Path Selection on $T_{\mathcal{H}}$ and G

For the implementation of the routing scheme in this system, all of the path sets $P_{u,v}$ are pre-computed by the master router, and then the appropriate sets $P_{u,v}$ are sent to all routers (but not hosts). The routing table is created in response to the master router broadcasting a *Create Routing Table* command. This process is detailed in Figure 11.

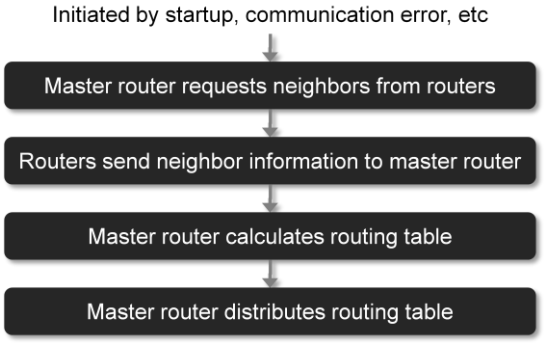


Figure 11: Routing Table Creation Process

4. CONCLUSION

This toolkit is still in development, so the effectiveness of the toolkit is not yet known. As of this writing, the topics discussed in Section 2 have been implemented and work according to design, although the topics discussed in Section 3 are mostly unimplemented thus far. After these elements have been implemented, this project will enter its next phase. During the next phase, the two software systems will be implemented to make developing parallel algorithms much easier. A subset of the Message Passing Interface (MPI) version 1 will be implemented as a middleware layer. MPI was chosen for its ubiquity in parallel processing on distributed systems such as server clusters. Much of the functionality of MPI won't be implemented because many of its features aren't relevant to an embedded environment. For example, there is no need for job management, because processing jobs on this system won't be initiated by humans.

A parallel compiler for SequenceL will be modified to run on the toolkit. Due to the nature of the language, it is easy for the compiler to find the parallelisms in the code automatically. This property means that programmers don't have to parallelize their code by hand, thereby reducing the development time \cite{sequencel}. The compiler will actually be a SequenceL to C compiler that makes use of MPI for the parallel code, which will make the code much more portable than a straight to assembly compiler. This approach also makes use of the decades of optimization that have gone into C compilers and allows easy integration of SequenceL code with embedded programming specific and even platform specific code.

After the toolkit is finished, a developer who wants to create a distributed parallel embedded system should be able to do so much more rapidly than was previously possible. From a hardware perspective, this toolkit will be just about as close to plug and play as one can get in an embedded system. From a software standpoint, one need only write the sequential algorithm in SequenceL, and the toolkit will take care of the details.

5. REFERENCES

[1] HyperTransport Consortium. "HyperTransport™ technology overview." referenced at **http://www.hypertransport.org/tech/index.cfm**, 2005.

[2] RapidIO® Trade Organization. "Rapidio® Technology overview." referenced at **http://www.rapidio.org/education/technology overview**, 2008.

[3] Phillips Semiconductor. "The I^2C-bus specification." referenced at

[4] R. Bosch. "Can Specification." referenced at **http://www.semiconductors.bosch.de/pdf/can2spec.pdf**, 1991.

[5] Harald Räcke. **Data Management and Routing in General Networks.** PhD thesis, Universität Paderborn, 2003. **http://www.nxp.com/acrobatdownload/literature/9398/39340011.pdf**, 2000.

[6] D. E. Cooke and J. N. Rushton. "Sequencel - an overview of a simple language." **In Proceedings of the 2005 International Conference on Programming Languages and Compilers**, pages 64–70, June 2005.