Amygdala-MP

(Multi Processor Extensions for Amygdala)

Rüdiger Koch rkoch@rkoch.org

March. 30, 2002 (version 0.2)

Abstract: Design issues and implementation of multiprocessor extensions are discussed as implemented in version 0.2 of Amygdala (SMP only). A short overview over the planned clustering design is given.

1 Intro

Running a neural network simulation parallel on multiple, potentially a very large number of CPUs appears very obvious. After all, the brain is considered a massive parallel computer by many researchers [Moravec, 2]. The processing power of a single silicon processor is much higher than that of an Amygdala neuron. This means that we want to run a group of neurons on each CPU. The groups have to be selected well from the entire neural network. Communication channels have to be set up for neurons in different groups to exchange spikes. The whole network needs to be synchronized because timing of incoming spikes crucial for learning and determining spike times.

Currently, only SMP is implemented. This includes the processors with hyperthreading and similar technologies (IBM Power4 and Intel Pentium 4 Xeon). Thus only the SMP design is discussed in depth. Running on clusters and other distributed memory architectures will be implemented soon. A design overview is given at the end of this paper.

There is some ambiguity about the word "network" in this paper. It can either mean a network of computing nodes and a neural network. We'll write NN for neural network and Net for a network of computing nodes.

Code elements such as class names are in a fixed font.

Table of Contents

2 Multithreading or Multitasking

Modern operating systems provide multitasking and multithreading. If the hardware has multiple CPUs in a shared memory architecture (SMP), the operating system tries to distribute the workload of different tasks (processes) and threads to different CPUs. So the aim is to run groups of neurons in different tasks or threads and let them communicate through appropriate interfaces. A design using threads or different processes each have their own advantages:

Threads	***Tasks (Processes)***
• Easy sync of NN (shared simulation time) • Because of efficient data exchange, partitiononing can be arbitrary • threads are more portable to non-Unix platforms	• Easy to debug • SMP and distributed memory systems can have the same semantic

We implement a multi-threaded model, mainly because the partitioning of the NN and multi-threading issues can be hidden from the user. This is of particular importance with the advent of multiprocessors-on-a-chip systems. In the near future, all new computers will benefit from parallel execution so we want this feature to be easy to use.

3 Threading and Object Model

When running an Amygdala NN on one CPU, all neurons are owned by one single object of class Network which organizes the communication of the Neuron objects and the simulation time. For multi-threading we run several MpNetwork objects each holding only a part of the complete NN. Each instance of a MpNetwork object has an object of class Instance associated with it. Since each partition is represented by an MpNetwork instance we call a NN partition an Amygdala instance. The reason for the Instance class to exist is to separate communication issues from NN issues. So an MpNetwork object only has to know about instance IDs, but does not know how to talk to them. This design makes it easy to extend the current implementation by clustering. All these objects are managed by a single Node object. (The name Node was chosen because the functionality of the Node class will later be extended to talk to other nodes of a cluster or MPP system.)

The handling of SpikeInput / SpikeOutput and Layer classes changed a little compared to running in non MP mode. Since each Network object holds a

SpikeInput and a SpikeOutput object it is possible to have several I/O objects. The SpikeInput object is replaced by an object of class MpSpikeInput for MpNetwork objects. The MpSpikeInput class is an important part of the inter-instance communication.

Currently, it is the user's responsibility to create MpNetwork objects and populate each MpNetwork with Neurons. Later, high level functions will be provided in class Node to take care of that so the user will not have to deal with MpNetwork objects if he doesn't require more control.

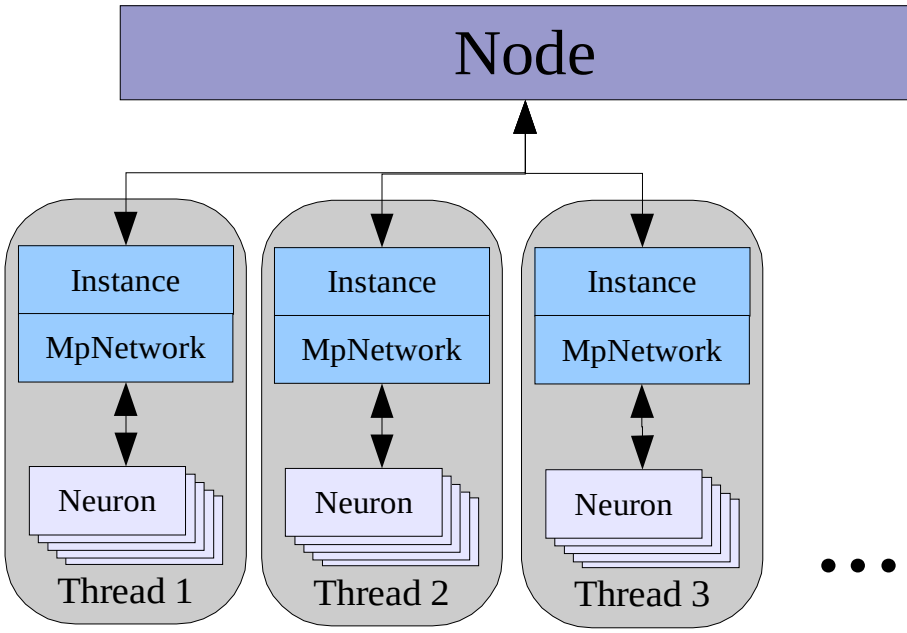

Fig. 1 shows a typical configuration. 4 Threads run under the control of a Node object. Each MpNetwork object has an associated Instance object.

3.1Synchronization

It is important to have a synchronized simulation time for all instances. To achieve that the simulation time (simTime) is a static member of the class Network which means that it is shared among all instances of MpNetwork. Incrementation is done after all instances delivered all spikes of a time step. One time step has 3 phases:

1. Instances run, delivering all spikes that are scheduled. When an instance runs out of spikes it goes to sleep.

2. The last running instance wakes up all the other instance so they can deliver spikes that came in since they went to sleep. Then it goes to sleep itself
3. The last running instance increments simTime and wakes up all other instances

3.2Exchanging Spikes between Instances

In summary, delivering spikes to remote instances is done as:

1. A spiking Neuron calls Node::sendSpike() for each Instance that contains receiving Neurons
2. The spike is transported to the remote Node if the instance is not local
3. The spike is delivered to MpSpikeInput
4. The spike is picked up by the receiving instance
5. The receiving instance resolves the receiving Neurons and the associated virtual NeuronID
6. Spike gets delivered to all Neurons

When a Neuron sends a spike, it gets delivered to all local postsynaptic Neurons. Then, the Neuron requests the Node to send each remote spike. Note that it is the receiving Instance's task to resolve which of it's Neurons will receive the spike. A Neuron only has information about which remote Instances have postsynaptic Neurons but knows nothing about the Neurons themselves. Also, the mode of transportation is chosen by Node.

The only mode of transportation implemented so far is calling the MpInputSpike::QueueSpike() method directly. This call adds the spike to the spikes vector until they are picked up by the receiving instance and delivered directly to Neuron::InputSpike() after resolving which Neuron's are addressed. Note that the spikes vector is the interface between two threads. The structure is therefore protected by a mutex.

The dendrite table of a Neuron contains the relation NeuronID<=>weight. Since NeuronID is not unique in the whole NN, but only within an instance, a virtual NeuronID is introduced which maps to a pair <instanceID, neuronID> of the sending instance. The virtual NeuronIDs are in the range 0xFF000000 to 0xFFFFFFFF. Real NeuronIDs may not be in this range when running multiple instances. The virtual NeuronID is relevant only within a Neuron. With this design, we avoid the need for two different dendrites, one for local and one for remote so we can keep the Neuron class almost free of MP specific code.

Resolving both virtual NeuronID and the receiving Neuron is done with the MpNetwork::AxonMap data structure:

A pair <virtual NeuronID, pointer_to_receiving_Neuron> is stored in a vector. This vector represents the subset of synapses the remote axon is connected with in this Instance. This remote axon is identified by the two hash_map structures

which use the remote InstanceID and the remote NeuronID as keys.

4 Performance

Several test were run in order to assess the potential of the current architecture. For the tests, the buildlayer example of the samples directory was used. All tests were carried out on a 2 processor SMP computer:

Mainboard: Tyan Thunder

Processor: 2xPentium II (Deschutes), 350 MHz, 512 KB cache

Memory: 128 MB on a 100 MHz bus

Kernel: Linux 2.4.2 (RH 7.1)

4.1 Running the original buildlayer program

The program was run on the SMP system. Since it is only one thread, only one CPU was used while the other was idle.

real	2m42.846s
user	2m42.830s
sys	0m0.100s

4.2 Running 2 buildlayer programs simultaneously

real	2m47.096s	2m46.931s
user	2m44.540s	2m44.420s
sys	0m0.120s	0m0.180s

Now both CPUs are utilized to 100%. The small slowdown compared to (1.) is likely due to memory I/O issues on the CPU bus. Also, other processes (xterm, X, kernel) cannot longer be shifted onto the idle CPU - they have to be pre-empted.

4.3 Running multiple buildlayer networks in synchronized threads

4.3.1 Running 2 NN in 2 threads:

real 3m12.887s

user 5m28.830s

sys 0m0.670s

4.3.2 Running 4 NNs in 4 threads

real 6m6.186s

user 11m3.860s

sys 0m1.600s

The threads do not exchange spikes in this example. Other tests (not documented here) show that the overhead of the spike exchange between different Amygdala instances on a SMP system is neglectable compared to the delivery of the spikes in the Neuron::InputSpike() function.

There are several factors for the slowdown. Most comes from synchronization. Threads that finish their 100us time step first have to wait until the last thread increments the simTime. It leaves the processors idle for about 14% of the time when running 2 threads. This idle time can be utilized by using more threads, though. The run with 4 threads suggest to run at least 2 times more threads than there are CPUs in the system, especially if the nets are not symmetric as in this example. Here, the idle time was reduced to less than 9%. Obviously there is room for further improvement here. The rest of the slowdown comes from the buildup of the NN which was not multi threaded. Buildup time thus doubled.

The test runs show excellent scalability on Intel SMP systems with 2 CPUs. Since the number of concurrent threads is not limited, the system should scale well on high end RISC systems.

It was assumed that Amygdala's bottleneck is memory I/O. This is obviously not the case since Amygdala scales so well on a SMP system with a weakness in memory I/O and strong in-cache execution.

5 Future extensions

SMP is rather limited. The largest SMP systems have around 100 CPUs and are rather expensive. Distributed memory systems are cheaper and more scalable. A

cluster with 64 workstations costs only a fraction of a 64 CPU server. MPP systems can scale very high. The strongest supercomputer, ASCI White, is such a system (IBM SP2). It runs on more than 8000 CPUs with a main memory of 6TBytes. IBM already announced Blue Gene, a massive parallel system with 1 Million CPUs and 1PetaByte of memory. If Moore's law holds, such systems should be available to the public within a few years. Blue Gene is notable in so far as it might be the first computer with memory and processing power roughly equal to that of the human brain [Moravec, 2]

It should be noted that unlike SMP the partitioning of the NN must be carefully chosen to minimize spikes delivered to remote nodes. How much depends on bandwidth and latency of the underlying network.

5.1 NOWs (Network Of Workstations)

NOWs communicate via TCP/IP. While it is possible to use a messaging system such as MPI or PVM it is much more efficient and also simpler to use sockets and UDP. The problem with MPI or PVM over TCP is not so much the additional overhead but the guaranteed sequence. If a packet is lost during transmission the kernel stores all data until the packet is successfully re-sent. This causes the hangs we all experience during Web surfing. Since a NN is very tolerant against noise we can afford the loss of spikes and even their duplication. The order of packages is completely irrelevant. Immediate delivery is needed, however.

It is crucial that all Nodes run synchronized. There is no need for hard synchronization, though – the Nodes don't need to have exactly the same simTime. It is advantageous to relax synchronization requirements as much as possible for performance reasons.

The Node spawns a listener thread to receive incoming messages and hand them off to the target thread(s) as fast as possible. Nodes are identified by their Node ID which in case of a IPv4 network is their **`struct sockaddr_in`** datatype, a 32 bit integer.

5.1.1Exchanging spikes

It was already mentioned that spike exchange uses UDP as transport protocol. The Node object is managing the exchange of spikes. The listener thread is then feeding incoming spike messages into the MpSpikeInput class in the same way as the other local threads. Sending spikes happens immediatelly. An Instance that wants to send a remote spike requests this from the Node.

5.1.2Synchronization of Nodes

Every Node decides by itself when to increment simTime. When a Node increments it's simTime it notifies all other Nodes with an [INCREMENT SIMTIME] multicast message. The decision of a Node to increment is based on 3

conditions:

1. The Node itself finished this timestep. No spikes are pending.
2. No spikes are to be expected for this timestep. This means that the Node **F**fastest is not too far ahead of the slowest Node. In this case **F** waits until it gets a notification from the slowest Node **S**lowest that it incremented or, since we are using UDP where packages can get lost, it waits until any other Node **X** passed it. To find out if the slowest Node is in fact a dead node A checks for incoming spike requests from **S**.
3. If the cluster is large it can be configured to continue even if Nodes are dead. In this case, Node **F** can send out a [IS NODE S DEAD?] request. If **S** does not react on this before the majority of all other Nodes confirmed their opinion that **S** is dead with [DEAD_NODE] messages, then **S** will be assumed dead and further messages from **S** will be ignored. **A** can increment simTime.

5.1.3Joining and leaving the cluster

Leaving a cluster requires no precautions on other Nodes. The other Nodes will notice that a Node is not active any more and will declare it dead with a majority decision. Spikes are no longer sent but [INCREMENT SIMTIME] messages still are so the node can sync when joining the cluster later again.

A dead Node may join a cluster again later by sending an [ALIVE AGAIN] message. It multicasts this message until it received a [ALIVE AGAIN ACK] message from all other Nodes. An [ALIVE AGAIN ACK] is followed by a list of all dead Nodes [DEAD_NODE] so the reincarnated Node can update it's database of dead Nodes. A reincarnated Node goes into operational mode when it received the [ALIVE AGAIN ACK] confirmation from a majority of all Nodes in the cluster.

5.1.4Starting and shutting down the cluster (controlling Node)

Starting and shutting down the cluster requires a controlling Node. A controlling Node will in most cases be a GUI application. Once the cluster is running the controlling Node is not required except for performing operations such as saving the Amygdala network or shutting down.

A cluster is started up by first running all Nodes with suitable means, either manually or via rsh, ssh or any cluster management tool. A Node first parses the command line options by calling the static function at the beginning of the main() function:

```
Node::Init(int &argc, char* argv[]);
```

When the function returns, all recognized command line options are stripped so the program can start doing it's own parsing of command line options. Then the Node loads the cluster configuration file. Command line options recognized are:

--node-id=12.34.56.78 This must be given in case the computer has multiple IP addresses. It must match one of them. The node will then bind to this IP address.

--config=http://foo.bar/config.xml This is mandatory

Nodes that are up are multicasting [STANDBY] messages to indicate their availability and the fact that they did not yet receive [STANDBY] messages from all other Nodes. Once a Node collected [STANDBY] messages from all other Nodes it starts sending [READY_TO_GO] messages to the controlling Node. It also starts receiving [SPIKE] messages. When the controlling Node has all [READY_TO_GO] messages it expects it multicasts [GO] messages until it received a [RUNNING] message from all Nodes.

5.1.5Loading and Saving

The cluster can only load a network when starting up by taking a command line parameter pointing to the cluster config file which contains the necessary data. This file contains the locations of the Network to load and save. Loading can either be local, via NFS or via HTTP by specifying the appropriate http:// or file:/ URL in the configuration file. Saving is always via file:/.

To Save an Amygdala network running on a cluster the cluster must be halted first. To achieve that the controlling Node sends out [STANDBY] messages. Every node that receveives such [STANDBY] message goes into standby itself. A Node that has received [STANDBY] or [READY_TO_GO] messages from all other Nodes goes into ready_to_go mode and sends [READY_TO_GO] messages. The controlling Node then multicasts [SAVE] messages until it receives a [STANDBY] or [READY_TO_GO] message from all Nodes. Once the controlling Node has received [READY_TO_GO] from all Nodes it may start the cluster again.

5.1.6The cluster configuration file

The cluster configuration file is needed to bootstrap the cluster. It's location is specified as a command line parameter and can be either a http:// or a file:/ URL. The format is XML conforming to the cluster.dtd file. It specifies:

- The multicast group
- Which node runs which Instances
- The locations from where to load the Amygdala files
- The locations where to save (optional)

5.1.7UDP based Protocol

Since the protocol includes integers they have to be ordered in network byte order, particularly the identifiers AmTimeInt and AmIdInt. There is no security built into the protocol. Every package received is trusted. Nodes are distinguished by their IP address. Multicast messages are marked.

Message type	Memory allocation

[ALIVE AGAIN]
(multicast)
	`struct alive_again{`
	`  char message_type;`
	`};`

[ALIVE AGAIN ACK]	`struct alive_again_ack{`
	`  char message_type;`
	`};`

[DEAD_NODE]	`struct dead_node{`
	`  char message_type;`
	`  struct sockaddr_in;`
	`};`

[GO]
(multicast)
	`struct is_node_dead {`
	`  char message_type;`
	`};`

[INCREMENT SIMTIME]
(multicast)
	`struct increment_simtime {`
	`  char message_type;`
	`  AmTimeInt NewTime;`
	`};`

[IS NODE S DEAD?]
(multicast)
	`struct is_node_dead {`
	`  char message_type;`
	`};`

[STANDBY]
(multicast)
	`struct standby {`
	`  char message_type;`
	`};`

[READY_TO_GO]	`struct ready_to_go {`
	`  char message_type;`
	`};`

[RUNNING]	`struct running {`
	`  char message_type;`
	`};`

[SHUTDOWN]
(multicast)
	`struct shutdown {`
	`  char message_type;`
	`};`

[SPIKE]	`struct spike {`
	`  char message_type;`
	`  AmIdInt SendingInstanceID;`
	`  AmTimeInt SpikeTime`
	`};`

6 More future extensions

6.1 Massive Parallel Machines

Machines of this class are very similar to NOWs. They also have different nodes with each node running a copy of the operating system. Their bus system is considerably faster and has much lower latency than an ethernet, however. If the same UDP messaging as for NOWs can be used or if MPI messaging is more suitable has to be investigated.

6.2 Scaling very high

A cluster as described is not scalable beyond a few dozen Nodes. To allow even higher scalability it is possible to use multiple multicast groups that are partly overlapping. Certain Nodes in such a cluster would belong to 2 or more multicast groups. A requirement would be that spikes are not sent beyond a multicast group. Such a multicast group can then be considered a compartment of a whole brain. The following image illustrates such an arrangement. Each multicast group is represented by a sphere of a certain color:

This image represents a simple variant. In a more complex arrangement it would be possible to have more connectivity of multicast groups than shown here, for instance overlapping the red group in the middle with the gray group at the bottom. To make such an arrangement effective, synchronization takes place only within any given multicast group and their direct neighbors.

7References

1. Dipl.-Ing. Cyprian Graßmann, Prof. Dr. Joachim Kaufmann: Distributed, Event Driven Simulation of Spiking Neural Networks. http://web.informatik.uni-bonn.de/II/ag-anlauf/spikelab/NC98.pdf
2. Moravec, Hans, Mind Children, Harvard University Press, 1988.