California Institute of Technology
Department of Computer Science
Computer Architecture

CS184b, Spring 2005 Assignment P5,P6,P7: Route Evaluation Monday, May 2

P5 Due: Monday, May 9, 9:00AM
P6 Due: Monday, May 16, 9:00AM
P7 Due: Monday, May 23, 9:00AM

Goals:

- Build Routers to assess routing time and requirements
- Compare Time Multiplexed and Packet Switched Routing

Collaboration: This assignment is a group assignment. A pair of students should be responsible for each of the two routers required.

Team: The class makes up 1 team = {hbarnor, nmehta, nachiket, mwilson}

Turnin: We have created directories: `/scratch/ic/cs184/project/{p5,p6,p7}`. Please put the files requested in that directory (but keep your a master copy elsewhere, that directory is not backed up). Email `mdel@cs.caltech.edu` when the materials are in the directory and ready for review.

P5 Tasks:

1. Develop infrastructure for a Time-Multiplexed Router; Provide your code for `TMRoute` including the routines `FindLeastCostPath` and `AllocatePath`.
2. Demonstrate the Time-Multiplexed Router can allocate one route between a source and sink node in the SmallConceptNet dataset on a (ChipZ,ChipY,ChipX,Y,X)=$4\times4\times4\times4\times4$, $W = 2$. Write out and turnin the route file for the completed route.
3. Develop infrastructure for a Packet-Switched Route Simulator; Provide:
   - `PacketSim`
   - `RouteFunction`
   - `PacketSimBlock`
   - `PacketSimSwitchBox`
4. Demonstrate the Packet-Switched Route Simulator can send a packet from one source to one sink in the SmallConceptNet dataset on a (ChipZ,ChipY,ChipX,Y,X)=$4\times4\times4\times4\times4$, $W = 2$. Turnin a cycle-by-cycle trace for the packet. (It is not necessary to have completely resolved all issues in the route function to assure deadlock avoidance in order to route a single packet; this gives you freedom to resolve route function issues over the first two weeks of the assignment.)

P6 Tasks:

1. Complete your time-multiplexed router; Provide your completed code for `TMRoute`.
2. Demonstrate the Time-Multiplexed Router can route the complete route task for the 100% activity SmallConceptNet on (ChipZ,ChipY,ChipX,Y,X)=$4\times4\times4\times4\times4$, $W = 2$.

   Write out and turnin the route file for the compeleted route.

   Report the worst-case latency for a single source-to-sink connection in the route. Report the total cycles taken to completely route the task set.
3. Finish defining packet-switched routing functions and installing them into packet-switched router; Provide final versions of:
   - `PacketSim`
   - `RouteFunction`
   - `PacketSimBlock`
   - `PacketSimSwitchBox`
4. Demonstrate the Packet-Switched Route Simulator can simulate a complete routing task 100% activity SmallConceptNet on (ChipZ,ChipY,ChipX,Y,X)=$4\times4\times4\times4\times4$, $W = 2$. Report the worst-case latency for a single source-to-sink connection in the route. Report the total cycles taken to completely route the task set.

P7 Tasks:

1. Run the time-multiplexed router against the following network route tasks:
   - ConceptNet full communication step for 512 FPGA design; (ChipZ,ChipY,ChipX,Y,X)=8×8×8×Y×X – you determine Y, X, and W to fit on a single XC2V-6000 and provide the highest performance (see below).
   - ConceptNet full communication step for 64 FPGA design; (ChipZ,ChipY,ChipX,Y,X)=1×8×8×Y×X – you determine Y, X, and W to fit on a single XC2V-6000 and provide the highest performance (see below).
   - SMVM communication for 128 FPGA design; (ChipZ,ChipY,ChipX,Y,X)=2×8×8×Y×X – you determine Y, X, and W to fit on a single XC2V-6000 and provide the highest performance (see below).

2. Run the packet-switched simulator against the following network route tasks:

   (a) ConceptNet communication step for 512 FPGA design; (ChipZ,ChipY,ChipX,Y,X)=8×8×8×Y×X – you determine Y, X, and W to fit on a single XC2V-6000 and provide the highest performance (see below).
   - full communication step
   - 50% active communication step represented by `large50.gph`
   - 30% active communication step represented by `large30.gph`
   - 20% active communication step represented by `large20.gph`
   - 10% active communication step represented by `large10.gph`
   - 5% active communication step represented by `large5.gph`

   (b) ConceptNet communication step for 64 FPGA design; (ChipZ,ChipY,ChipX,Y,X)=1×8×8×Y×X – you determine Y, X, and W to fit on a single XC2V-6000 and provide the highest performance (see below).
   - full communication step
   - 50% active communication step represented by `large50.gph`
   - 30% active communication step represented by `large30.gph`
   - 20% active communication step represented by `large20.gph`
   - 10% active communication step represented by `large10.gph`
   - 5% active communication step represented by `large5.gph`

   (c) SMVM communication for 128 FPGA design; (ChipZ,ChipY,ChipX,Y,X)=2×8×8×Y×X – you determine Y, X, and W to fit on a single XC2V-6000 and provide the highest performance (see below).

   Note, in P4 you reported feasible contours for (N, W) and (N, W, T). If these contours are non-trivial, it may be necessary to explore multiple points to find the one that reduces the communication steps (for both Parts 1 and 2 here). Since T is an output of the time-multiplexed route, it may be necessary to iteratively refine your design until you reach a consistent T. Provide graphs justifying the final (N, W, T) you select.

3. For each of the 3 cases, plot communication time versus percent activity for the packet-switched case along with time-multiplexed communication time (flat line).

4. For each of the three cases, identify the worst-case route latency. Generate another plot which includes this along with total communication time.

1 Routing Infrastructure

In this section we give an overview of some of the classes which you will build upon. This is not an exhaustive list. See the Javadoc for the classes to see full interfaces. This section draws your attention to some of the functions which will be most helpful in building your router and simulator.

1.1 `GraphMachine` Classes

We provide a package `GraphMachine` which contains the following classes to support your router and simulator designs:

- `GraphMachine` – the topology and graph of a full physical network containing GraphMachineNodes (PEs and Switches) and links between them
- GraphMachineNode – generic interface which represents physical nodes in the physical graph, generalizing both GraphMachineBlocks and GraphMachineSwitchBoxes
- GraphMachineBlock – represents a PE and implements the GraphMachineNode interface
- GraphMachineSwitchBox – represents a Switch Box and implements the GraphMachineNode interface
- GraphMachineChipBorder – represents a special Switch Box to handle chip to chip traffic
- GraphMachineWire – general directional wire that connects physical GraphMachineNodes
- GraphMachineWirePhase – extends GraphMachineWire, keeping track of its phase number and dealing with the fact that it connects to wires in different phases. Basically, this is a container to hold the state for a single phase of a time-muxed wire.
- GraphMachineWireTM – extension of GraphMachineWire to support time slots on Time-Multiplexed wires. This is composed of GraphMachineWirePhases.
- GraphMachinePath – for static routing, this contains the path for a single source to sink rink; use with your TMRoute to backtrace shortest paths and good for reading/writing. You will use GraphMachinePath to store allocated routes, review routes for congestion, rip-up routes for rerouting, and to write out a record of routes into an ASCII test file (for final reporting and for debugging).
- GraphMachineEdge – represents a physical source-sink pair associated with a logic edge in the base graph.

1.2 Other Classes

The logical graph is represented as a `MultiWeightGraph`. `MultiWeightGraph` has its own package. The logical nodes are MultiWeightGraphNodes. Each GraphMachineBlock will contain a collection of MultiWeightGraphNodes. To find the successors of a MultiWeight-GraphNode, you can use getNext(output,fanout). To find out the number of outputs a node has, use getNumOutputs(). To find out how many fanouts a particular output has use get-FanoutCount(whichoutput). Once you have the successor node, you may need to know its destination location. You can do this using Topology.getPhysical(MultiWeightGraphNode).

1.3 Things you will want to do

- *Get a list of GraphMachineEdges that need to be routed. The router/simulator needs to have this list so it knows what it needs to route.*
  Topology.getEdges() will return a collection of GraphMachineEdge.
- *Find the set of wires that connect out of a GraphMachineSwitchBox. The time-multiplexed router will want to perform a search along each of the output directions.*
  GraphMachineSwitchBox.getOutputs() will return all the output wires. GraphMa-chineSwitchBox.getOutputs(Wire) will return output wires reachable by the given in-put wire.
- *Find the GraphMachineNode at the other end of a Wire. When traversing between nodes in either the router or the simulator, this is how you discover which GraphMa-chineNode (GraphMachineSwitchBox or GraphMachineBlock) is next.*
  Wire.getSink() will give you the GraphMachineNode at the far end of the wire.
- *Find the type and location of a node. This will tell you where you are in the network. For the Time-Multiplexed router, you want to know if you have reached the destina-tion. For the Packet-Switched router, you want to know where you are so you can decide where to go next.*
  You must use the Java `instanceof` to find out if the GraphMachineNode is a Graph-MachineBlock or a Switch. If you have the destination as an object, you can compare this node to your destination. If you have the destination as an address, you can get the ordinates of the address using accessors getChipZ(), getChipY(), getChipX(), getX(), getY().

1.4 Some Highlights of `GraphMachine` Class Functionallity

GraphMachineWire represents a wire connecting two nodes. There is a slot (came_from_wire) where you can store the predecessor wire when you are allocating a route path; this may ease route tracing. There is also state for holding the set of paths sharing this wire. Two edges that have the same source can often share a wire without causing congestion. During PathFinder routing, it is useful to allow wire sharing while negotiating congestion. Important functions you will need:

- int getDirection() – get the direction a wire is pointing. A SwitchBox calculating delay based on direction of inputs and outputs will need this.
- int getHist() – get the congestion history value for this wire.
- void setHist(int) – update the congestion history value for this wire.
- void addPath(GraphMachinePath p) – this allocates this wire for path p.
- void removePath(GraphMachinePath p) – this removes path p from the wire.
- int getCongestion(GraphMachinePath p) – this reports the congestion contribution of this wire to the path if path p is added.
- void setCost(int) – annotate the wire with the cost computed by the router cost function.
- int getCost() – retreive cost of wire.
- resetTempCost() – reset temporary routing state of a wire. Does not upset occupants or history.
- Wire.resetAllTempCosts() – reset all Wire
- int Wire.examined – a field to hold a path search id. Useful for tagging wire state so that each wire need not be reset between each path search.

GraphMachineWireTM A multiphase wire is a wire that has separate time slots (and hence separate path users) on each time step. This allows you to construct a multi-phased wire, but then treat it as a normal GraphMachineWire. GraphMachineSwitchBlocks automatically take care of connecting you from a given phase to phase+delay, where delay is the delay through the switch box. No harm will come from using a 0 phased GraphMachineWireTM as a normal GraphMachineWire, but that would just add area and time overhead for no benefit, so don't do that.

Topology A Topology represents the physical network of PEs connected by Wires and Switches. Useful functions:

- getNodes() – returns all the nodes (MultiWeightGraphNode)
- getBlocks() – returns all the blocks in the Topology. Useful for walking over the PEs.
- getPhysical(MultiWeightGraphNode) – get the PE (Block) which holds the given logical node.
- mapNodes(GraphMachineNodeFunction f) – applies f.execute() to all nodes in the Topology. This is useful for performing ProcessInputs and AdvancedOutputs (see Packet-Switched Simulator design) on all nodes in the graph.
- mapWires(GraphMachineWireFunction f) – applies f.execute() to all wires in the Topology. This is useful for performing HistoryUpdate or clearing state (See Time-Multiplexed Router Design) on all wires and phases in the physical graph.
- unroute() – rips up all paths on all wires.
- getEdges() – returns a collection of GraphMachineEdge for all edges in the system. Hint: you might want to use this for Time-Multiplexed routing
- getEdges(MultiWeightGraph) – return a collection of GraphMachineEdge which are the subset of the system specified by the input graph. Useful for specifying a subset of edges to route. Hint: You might find this usefull for Packet Switched routing

GraphMachinePath This represents a source→sink path in the Physical Graph Machine.

- dump() – rips up the entire path
- canShare(GraphMachinePath p) – true if this path does not conflict. false if the two edges conflict.

You need to provide:

- AllocatePath(Wire w) – This should trace back from the last wire to the source and allocate each wire.

You will need to extend the base GraphMachinePath class included with icppr. Set your class name in the technology file:

```
(override_class edu.caltech.icppr.GraphMachine.GraphMachinePath
edu.caltech.icppr.GraphMachine.MyPath)
```

See the Technology section (Section 10) for more information.

GraphMachineNode is the generic interface for GraphMachineBlock and GraphMachineSwitchBox. It provides:

- Wire [] getOutputs() – outputs from a node.
- Wire [] getOutputs(Wire) – outputs reachable for the specified input wire. This will be a zero length array when there are no wires or the node is not a routing component (PEs, things that don't directly forward messages from their inputs to their outputs).
- int getChipX() – get the X position of the chip the node is in
- int getChipY()
- int getChipZ()
- int getX() – position in the chip
- int getY()
- int getC() – for items that are replicated to make wider channels this is the specific channel. Among other things, this is likely to show up in printed routes.

TMRouteCostFunction.cost(GraphMachineEdge,GraphMachineWire) What is the cost of adding this edge to this wire? This is provided for use with PathFinder routing.

$$\begin{aligned} & cost(Wire, Path, old\_cost) \\ = \;& old\_cost + wire\_base\_cost \\ & + congestion\_coeficient \times (new\_occupancy\_of\_wire - 1) \end{aligned} \tag{1}$$

This class can also be overriden in the same fashion as GraphMachinePath. See the Technology description for more information.

2 Partition and Place Tasks to PEs

Since the number of PEs varies with T, W, and the size of your switch implementations, we cannot provide you with a static set of pre-placed files. Instead, you will need to partition the graphs for the target number of PEs (N_{PE}) in your system. As noted in the tasks, you may need to explore N_{PE},W,T tradeoffs within a fixed number of FPGAs. The partition/placer takes in a graph and a technology file and produces a placement file. When working on a low-activity ConceptNet task set, you should still place the full graph. Among other things, this allows you to use the same placement for all the different activity levels.

Usage:

```
icrun.sh GraphMachine.pp \
        -O <outputdir> -t <tech file> [-s <stats file>] \
        -max_imbalance \
        -place <placefile> \
        <netlist>
```

This top-level routine parses a command line and includes code to read in a technology file, read in a graph and place the graph onto the net.

- **netlist** – The topology will take in a netlist in the format of a MultiWeightGraph. MultiWeightGraph will basically read the files in the given format (*.mtx or *.gph) and build itself. The netlist specifying the static graph is a required argument.
- **placefile** – output file where the placement is written (See Section 11).
- **tech file** – The technology file specifies the network topology and network component properties (Section 10). It also specifies the size of the 5 dimensional array. This is required.
- **stats file** – Each driver may output a statistics file.
- **outputdir** – `-O <outputdir>` specifies the output directory.
- **max_imbalance** – The partitioner is given a number greater than 1.0 to specify the ratio of the maximum PE load to the average PE load. A reasonable value is 1.2.

$$\begin{aligned} N_{PE} &= ChipZ \times ChipY \times ChipX \times Y \times X && (2) \\ MeanEdgesPerPE &= \frac{\text{Total Edges}}{N_{PE}} && (3) \\ MaxEdgesPerPE &= (\textbf{max\_imbalance}) \times MeanEdgesPerPE && (4) \end{aligned}$$

3 Drivers

We provide you with the basic drivers (top level routines, like `main.c`):

- `gmtmr` – runs time multiplexed routing
- `gmprs` – packet routed simulation

These top-level routines parse a command line and include code to read in a technology file, read in a graph or placement, and invoke the appropriate router or simulator code. With these drivers in place, you should be able to focus on writing your router/simulator.

usage:

```
icrun.sh GraphMachine.gmtmr -place <placefile>  \
        -O <outputdir> -t <tech file> [-s <stats file>]
        [ -phases ## [-fixed_phases <bool>]] \
        -place <placefile> -route <routefile> \
        <netlist>

icrun.sh GraphMachine.gmpsr -place <placefile>  \
        -O <outputdir> -t <tech file> [-s <stats file>] \
        -messages <message netlist> \
        -place <placefile> -route <routefile> \
        <netlist>
```

Note example command lines for each driver can be found in the examples directory. See the README and Makefile.
`/cs/courses/cs184/spring2005/examples/p567`

Inputs and outputs are:

- **netlist** – The topology will take in a netlist in the format of a MultiWeightGraph. MultiWeightGraph will basically read the files in the given format (*.mtx or *.gph) and build itself. The netlist specifying the static graph is a required argument. The optional argument to `gmpsr` passed by `-messages` specifies the sparsely active subgraph.

- **placefile** – The full GraphMachine placement is a 5 dimensional array of logical elements. The placement should specify a list of logical nodes that go into each PE (See Section 11). The PEs are arranged as a 2d array on each GraphMachineChip. The full system contains a 3d array of those chips. `-place <placefile>` specifies the placement file.

- **tech file** – The technology file specifies the network topology and network component properties (Section 10). This is required.

- **stats file** – Each driver may output a statistics file.

- **outputdir** – `-O <outputdir>` specifies the output directory.
- **routefile** – Route drivers output a route file.

4 How to Use ICPPR

4.1 Where to find more detailed directions

We have some useful resources for working with icppr and specific directions for working on this project. You still should take a quick look at the howto in this document, as not everything is covered in the web howto.

- `http://www.cs.caltech.edu/research/ic/icppr/overview-summary.html`

  – the general icppr startup instructions. warning, might be a bit dated.

- `file:///cs/courses/cs184/spring2005/assign/p5/howto_icppr.html`

  – a more up-to-date set of instructions for:

  – getting icppr
  – configuring eclipse
  – creating a new project for your work
  – adding your project to a cvs repository
  – running from within eclipse
  – running from the command line

- `http://www.cs.caltech.edu/research/ic/icppr/`

  – general prebuilt icppr javadoc website.

4.2 Software

On the cs cluster it should be sufficient to run “eclipse”, or if that fails, `/cs/research/ic/software/eclipse`. And you can ignore this section.

Eclipse is our recommended platform for working with icppr. As such you will need:

- java – a java runtime and compiler. I suggest getting a current version of Sun’s jdk from http://java.sun.com (you will be wanting the jdk, not just the jre).
- eclipse – get it from <`http://www.eclipse.org`>

The other dependencies for the project are included either as part of the above or are grabbed with the icppr tree. Java 1.5 or later is required for working with icppr. But please don’t add any 1.5 specific syntax to the code base (1.5 classes are ok; *e.g.*: java.util.PriorityQueue).

4.3 Eclipse

Before getting icppr, you should set the eclipse java compiler to full java 1.4 support. If you don't want to make that change global, you can do so for the project, once its downloaded. Either way, it is required to build icppr.
Also, its suggested you disable two of eclipse's more paranoid checks so more important warnings are buried. Set "unused imports" and "non-static access to static" to "ignore".

4.4 Getting icppr

You should use cvs to grab icppr. From within eclipse, use the cvs repository perspective. The host is cvs.cs.caltech.edu, and the path is /cs/research/ic/cvsroot.
For command line cvs, use:
`CVS_RSH=ssh and CVSROOT=:ext:cvs.cs.caltech.edu:/cs/research/ic/cvsroot`
Its strongly recommended that you do this from within eclipse, as it will track things and provide a very nice frontend for cvs. Directions are more elaborate, including screen shots in the above mentioned howto.

4.5 Where to put your code

Create a new project. As you do so, make sure icppr is a dependency.
New classes should still be added to the package `edu.caltech.icppr.GraphMachine`, unless you have a specific reason to put them elsewhere.

4.6 Overriding classes

You might wish to override certain classes. GraphMachinePath is one such class, but there maybe more. The list should make it to the GraphMachine overview at some point.
To specify a default override, add a line to the Technology file: `(override_class BaseClass YourClass)`, as seen earlier in the GraphMachinePath description.
This will not interfere with normal created objects. But it will be used to determine the class when a new object is created with a factory constructor such as BaseClass.newBaseClass(args). This should leave the code looking pretty normal.
Instead of:

```
GraphMachinePath p=new GraphMachinePath(source, sink);
```

One would use:

```
GraphMachinePath p=GraphMachinePath.newGraphMachinePath(source,sink);
```

Just remember, this lets icppr transparently support your replacement classes. The factory will only have methods for prespecified constructors, and will not handle constructors with different inputs you add to your class. Also if you're instantiating your objects, you don't need to use the factories. A very simple example is available upon request (`rafi@cs.caltech.edu`) for those wanting to see a simple demonstration of how this works.

4.7 Running

In eclipse, there is a "run" item in the "run" menu. It will let you configure a "run". Which is basically a persistent environment and command line. It also has options to put the run in the "run" and "debug" quick lists. If you want to make the run available to others in your group, select "shared" (see the howto for a screen shot).

Running from the command line.

- Add your project directory to ICRUN_PREPEND_CLASSPATH.
- Make sure you have a java runtime available in your path
- run: icrun.sh <CLASSNAME> <ARGUMENTS> – <JVM ARGS>

Note, `icrun.sh` adds `edu.caltech.icppr.` to the specified classname, so leave that part out.
Also, if you wish to package up your work into a jar:
`cd <project dir> ; jar cf my.jar edu`
If you wish to run some jobs, while continuing to develop the source, make a jar and put it with your jobs and use that jar in the ICRUN_PREPEND_CLASSPATH.

```
cd ~rafi/eclipse/my_new_project
jar cf /tmp/silly.jar edu
cd /tmp
export ICRUN_PREPEND_CLASSPATH=/tmp/silly.jar
~/rafi/eclipse/icppr/dist/scripts/icrun.sh GraphMachine.SillyClass "say hi"
```

5 Time-Multiplexed Router

You router class is: `TMRoute`. TMRoute will take as an argument a `Topology` which it will route.

We provide a stub for TMRoute for you in:

- `/cs/courses/cs184/spring2005/assign/p5/TMRoute.java`

We suggest TMRoute be decomposed into the following pieces:

- Central Route Loop – loops over all point-to-point connections (GraphMachineEdge), rerouting them until they are all successfully routed (free of congestion)
- FindLeastCostPath – subroutine that finds the least cost path for a single point-to-point connection (GraphMachineEdge)
- AllocatePath – subroutine that installs a path once one has been found
- HistoryUpdate – routine to update the congestion history of each wire

The `Topology` will allow you to get a collection of edges (needed routes) using Topology.getEdges(). Path.dump() will allow you to rip up old routes, as you may need to do when you have congested routes. We are providing a default `TMRouteCostFunction` for you to use with your FindLeastCostPath. We recommend you use a *Pathfinder*-based routing algorithm [1]. Topology.mapWires() will help you implement HistoryUpdate.

6 Packet Switched Simulator

Our suggested decompositon for the Packet Simulator includes the following classes:

- PacketSim(Topology) – top level simulator; takes a Topology as an argument.
- RouteFunction – a function that looks at position in network and destination and directs the packet
- Packet – data structure representing a single packet
- Flit – data structure representing one network-width word of a packet
- PacketSimBlock – an extension of GraphMachineBlock (or a wrapper around it) for packet switch simulation
- PacketSimSwitchBlox – an extension of GraphMachineSwitchBox (or a wrapper around it) for packet switch simulation

PacketSim should include:

- Init() – clears packet queues, sets up source queues.
- boolean RouteStep(timestep) – performs one clock cycle; return value indicates if routing has completed (does any GraphMachineBlock or GraphMachineSwitchBox still have packets in its queues); this mostly calls ProcessInputs and AdvanceOutputs.
- ProcessInputs(timestep) – map over all GraphMachineBlocks and GraphMachineSwitchBoxes and compute what happens on this cycle.
- boolean AdvanceOutputs(timestep) – move things to outputs; this is the second phase of the RouteStep. Two phase architecture makes order of block/switch processing unimportant.

PacketSimBlock and PacketSimSwitchBlocks will need to include the queues for packet simulation. PacketSimBlock simulates the source, placing packets (flits) into network and consuming packets from the net. PacketSimSwitchBlock moves flits from queue to queue. These include:

- Init() – clear packet queues.
- ProcessInputs(timestep) – uses RouteFunction to make decisions; advances flits from queue to queue; this should take care of modelling appropriate number of clock cycles through a SwitchBox.
- boolean AdvanceOutputs(timestep) – actually moves packet to next queue as appropriate.

You may need to configure your switches with parameters for the number of cycles they require to traverse. You will also need to configure the size of your queues. The Technology File (Section 10) provides a standard way for you to pass these parameters into the simulation.

RouteFunction(myX, myY, myChipX, myChipY, myChipZ, targetX, targetY, targetChipX, targetChipY, targetChipZ) – function that decides to which output a packet should be routed. We're suggesting this be isolated into a separate class so that it can be changed independently of the simulator structure.

Packet includes:

- destination (header)
- payload
- timestep for entering net – the sink node can look at this to record the latency for a packet; note that we ask you to report the maximum transit latency for a packet. In general, packet latency is an important performance metric to quantify.

Flit represents a single, network-width word of a packet. This is what actually sits in each queue slot and advances on each cycle.

- Packet – pointer to the packet
- which – integer indicating which flit

7 Routing Tasks

Task sets will be given which are independent of the particular partitioning of graph nodes to PEs. Consequently, sometimes you will need to route a message from a PE back to itself. Assume the PE can route self messages in one cycle. Assuming the PE can process each incoming/outgoing message in one cycle, this also accounts for the processing cost associate with the local work on each node.

We provide you with the following route task sets:

For P5 and P6 look in: `/cs/courses/cs184/spring2005/assign/p5/benchmarks`

- sparse matrix: `test2.mtx`
- SmallConceptNet graph: `matter-basic-small-lim128.gph`
- SmallConceptNet 100% activity: `smallnet100.task`
- SmallConceptNet 10% activity: `smallnet10.task`

For P7 look in: `/cs/courses/cs184/spring2005/assign/p7/benchmarks/`

- sparse matrix: `fidapm37.mtx`
- StandardConceptNet graph: `matter-basic-default-lim128.gph`
- StandardConceptNet 100% activity: `largenet100.task`
- StandardConceptNet 50% activity: `largenet50.task`
- StandardConceptNet 30% activity: `largenet30.task`
- StandardConceptNet 20% activity: `largenet20.task`
- StandardConceptNet 10% activity: `largenet10.task`
- StandardConceptNet 5% activity: `largenet5.task`

8 Route Functions

To avoid deadlock in the packet-switched routing case, we will need to restrict packet routing. Note that none of this applies to Time-Multiplexed routing; since Time-Multiplexed routing is coordinated globally, offline, the schedule can always avoid deadlock. Time-Multiplexed routing should use whatever routes give the least time-cycles required for routing the task set.

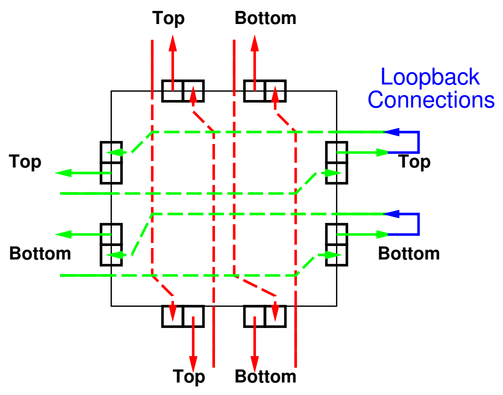
The simplest model would be to use dimension-ordered routing. You may choose to simply implement dimension-ordered routing throughout to keep the routing simple. As described above, the route function is a separate/isolated class (`RouteFunction`) and it should be easy to replace the routing function with a more sophisticated one for experimentation at a later date.

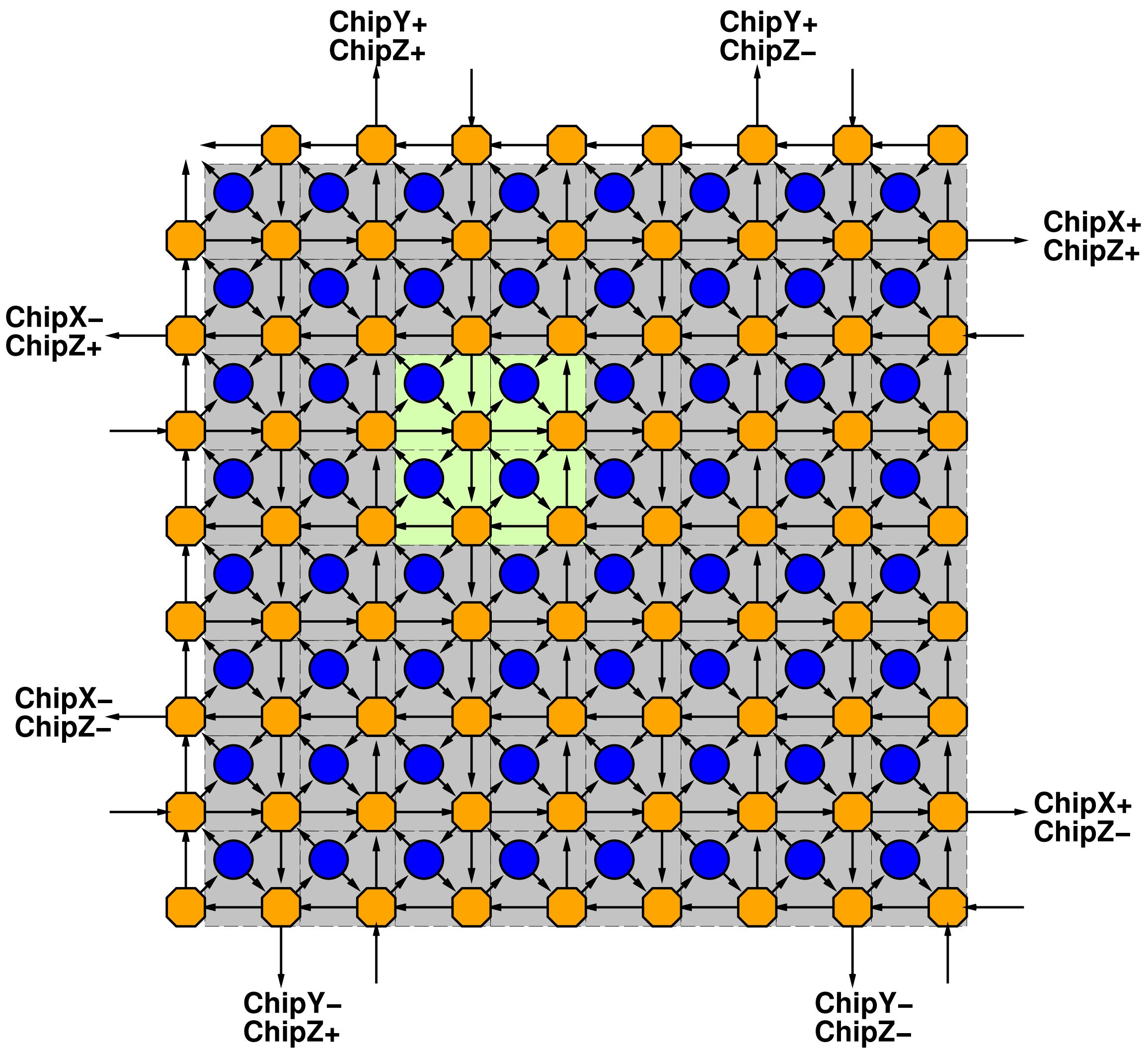
ChipZ routes also change ChipX or ChipY position, whereas it is possible to route in ChipX or ChipY without changing the ChipZ plane. Consequently, we should route in ChipZ first. After ChipZ is resolve, routing can proceed along ChipY, ChipX, Y, X.

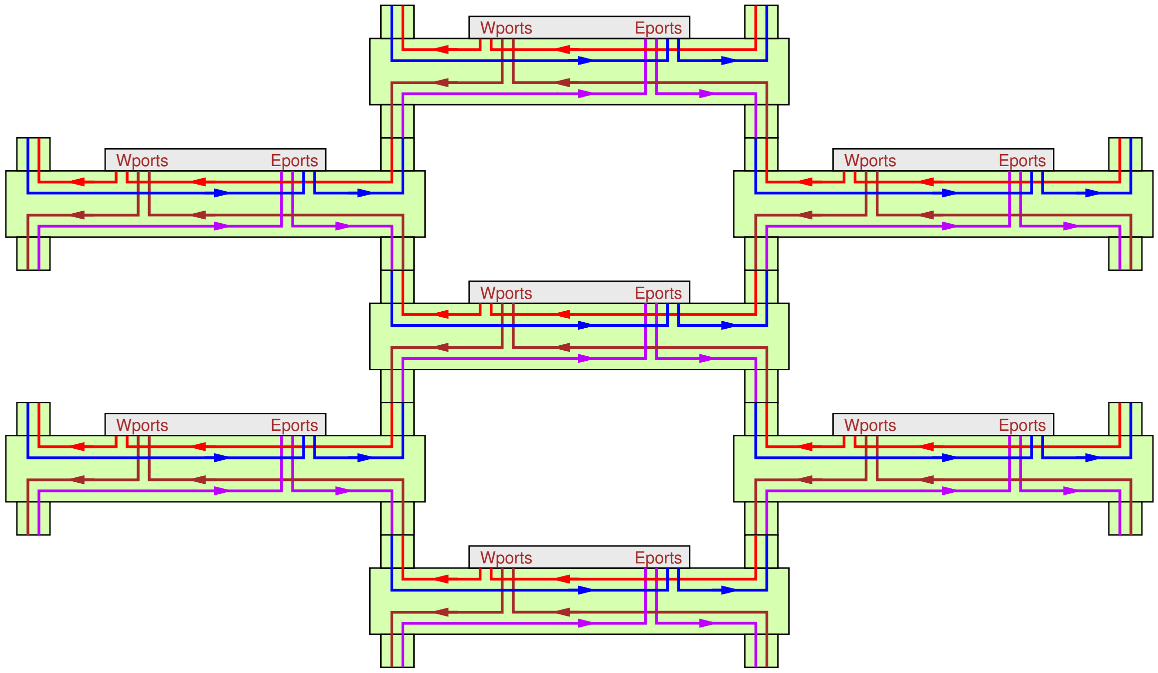
Note that IOs have been placed so that continuing in the same direction requires the traversal of only two switches. However, ports that change the Chip route dimension are not so close and will generally be expensive. Consequently, dimension-ordered routing at the ChipZ, ChipY, ChipX level may be the most efficient in any case. We could allow adaptivity in switching to ChipY versus ChipX; however, since the ports are far apart on the chip, it is unlikely we will have good feedback to decide which way to go. This suggests:

1. Exit appropriate ChipZX or ChipZY
2. We may overshoot the ChipX or ChipY target coordinate while routing in ChipZ; nonetheless it is probably best to keep routing along the same ChipZ connection until we achieve the desired ChipZ plane. This means we may hit the edge of machine during an overshoot. As noted below, we are suggesting loopback edge connections that deal with this boundary condition without further complicating the route functions.
3. Switch to ChipY
4. Switch to ChipX
5. Route West-first once enter the correct chip. (This is one place we may want to do non-dimension ordered routing; doing West-first instead of dimension-ordered is an option.)

The need to add special logic to deal with the edge of network condition would be troubling. To avoid that we add a loopback connection on the otherwise unused edge connectors. This way, we physically flip routing in the non-ChipZ dimension along which we have been routing. For example, the IO loopback shown below illustrates how this would be wired for a chip on the ChipX+ edge of the network. Once we hit the edge of the net, the we re-enter the chip and line up with ChipX- routing. Since we have certainly overshot the ChipX target coordinate, this heads us back toward the desired ChipX while continuing the ChipZ routing.


Top
Bottom
Loopback
Connections
Top
Top
Bottom
Bottom
Top
Bottom


ChipY+
ChipZ+
ChipY+
ChipZ–
ChipX+
ChipZ+
ChipX–
ChipZ+
ChipX–
ChipZ–
ChipX+
ChipZ–
ChipY–
ChipZ+
ChipY–
ChipZ–


Wports
Eports
Wports
Eports
Wports
Eports
Wports
Eports
Wports
Eports
Wports
Eports
Wports
Eports

9 Route Ordering

In the GMI PCE design at the end of CS184a, you noted that you could not have back-to-back messages for the same ConceptNet graph node without forcing a stall to avoid a WAR hazard in the pipelined memory access. Further, it may be beneficial for performance to distribute the messages you send rather than sending sequential messages to the same physical node or even the same output port direction on chip exit.

For the time-multiplexed router, you can schedule edge transmissions in the order you can route them and they can be consumed without stalling. Consequently, the order of message transmission and reception is part of what you are scheduling when computing the route.

For the packet-switched router, you need to keep the decisions simple. Most likely, you would go through the edges for a node in order when you send out message. The one freedom you do have is the ordering of edges within a node. You may consider a mapping pass that reorders the edges of a graph node once you know where graph nodes have been placed on physical PEs. You are not required to perform this reordering.

10 Technology

10.1 Dishoom.tech

This is where you put global configuration options, such as:

- Class of the Topology
- array dimensions in terms of chips
- starting number of phases
- switch box delays
- c

```
/* comment */
(system_wide_tech
     (topology edu.caltech.icppr.graph_machine.GraphMachine)
     (dimensions
          chip_x
          chip_y
          chip_z
     )
     (c ##)
     (phases ##)
     (message_serialization "")
     (switch_box_path_delays [array of integers])
)
```

10.2 Chip.tech

This is where you describe parameters that are specific to the on-chip topology

- Class of chip Topology
- array dimensions in terms of PEs on the chip
- switch box delays (will overide the global if specified here)
- c (will overide the global if specified here)

```
(chip_tech
        (topology edu.caltech.icppr.graph_machine.GraphMachineChip)
        (dimensions
               x
               y
        )
)
```

We will provide technology files for the parameters stated in the problem set. But you will need to fill in any switch box delays. Look in:

- `/cs/courses/cs184/spring2005/assign/p5/tech`

11 Placement File

The placement file is a simple list of nodes and their associated location. Each line contains a graph node name followed by six integers. These integers are the ChipZ, ChipY, ChipX, Y, X, and slot location for the graph node. slot is simply an offset within the PE. The ordering is not currently significant; we simply need to place each graph node in a single location in a different slot.

The example generates the placement file:
`example_8x8x8x8x8_c_2.matter-basic-small.place`

References

[1] Larry McMurchie and Carl Ebling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM, February 1995. <`http://www.cs.washington.edu/research/projects/lis/www/papers/postscript/mcmurchie-FPGA95.ps`>.