Routing in a hyper-cube


1 Routing in a Hyper-Cube

For this assignment, you must write a program that simulates a hyper cube of dimension $d$. The program should take in 2 arguments: the number of messages each node should send as well as the dimension of the hyper-cube (i.e., $d$).

Let us first look at the rules for routing in the hyper-cube.

1.1 Routing Rules

Remember, in a 1-dimensional hyper-cube we have two nodes; in a 2-dimensional cube we have 4 etc. In other words, a $d$-dimensional hyper-cube has $2^d$ nodes. The id of node $i$ is best described by its binary representation. Let us consider 2 nodes in a 2-dimensional hyper-cube: 00 (node 0) and 11 (node 3) (See figure 1). Since 00 is not directly connected to 11, the message must be routed through either 10 or 01 (either way would be fine), but for this assignment let us route left/right first and then up/down, so the right-most digit represents the left/right dimension and the left-most the up/down dimension. That is, we route from least to most significant bit.

![Figure 1: 1, 2 and 3 dimensional hyper-cubes.](image)

The complete route the message takes including the source and the destination is therefore: $00 \rightarrow 01 \rightarrow 11$.

In order to determine how many hops the message needs to go through (not including the source or the destination) can be obtained by counting the number of 1s in the result we get from computing the exclusive or between the source and destination and subtracting one. Indeed, the exclusive or result is the routing map to getting the message from the source to the destination:

$$00 \oplus 11 = 11$$

so we see that we have to route in both directions to get to the destination. Basically, the exclusive or of the source and the destination tells us in which
directions we need to route the message, a 0 in the right most position of the exclusive or tell us that the no routing is needed in the dimension represented by this bit.

Let us look at a bigger example: how do we route a message from 1101 to 1010 in a 4-dimensional cube? We start by computing the first routing pattern:

\[ 1101 \oplus 1010 = 0111 \]

which shows us that the destination node is 2 hops away. Since we route from least to most significant bit, we need to locate the first 1 searching the bit string 0111 from right to left. We find a 1 in the very first (the rightmost position), which corresponds to the number 2^0 (0001), which we can now use to compute the next hop. We do that by computing the exclusive or between this new pattern and the number of the node in which the message currently resides (in this case 1101):

\[ 0001 \oplus 1101 = 1100 = 12_{10} \]

We see that the message should be sent from 1101 to 1100, which we know to be true because we must route along the dimension represented by the rightmost bit, and when doing so, we flip the bit and arrive at 1100. Now the message is at 1100 and we can perform the computation over again:

\[ 1100 \oplus 1010 = 0110 \]

which should not be a big surprise (it is the same pattern as we got before with the last bit flipped as we already routed along that dimension). The first non-zero bit searching from right again represents the value 2. So we compute the exclusive or of 0010 and the node’s id:

\[ 0010 \oplus 1100 = 1110 = 14_{10} \]

One more time (this time at node 1110) we compute the exclusive or of the nodes id and the destination’s id:

\[ 1110 \oplus 1010 = 0100 \]

and search for the first non-zero value from right to left and find the 1 representing the value 4; again we compute the exclusive or of this number (4) and the node id in which the message resides:

\[ 0100 \oplus 1110 = 1010 = 10_{10} \]

which is indeed the destination, so once node 1110 delivers the message to 1010 it has reached its destination. I have written this little helper function:

```c
const unsigned int power2[12] = {1,2,4,8,16,32,64,128,256,512,1024,2048};

unsigned int compute_next_dest(unsigned int rank, unsigned int dest) { 
  if (rank == dest) // if we are already there
```
return dest;
unsigned int m = rank ^ dest;
int c = 0;
// find the first 1 searching from right to left.
while (m%2 == 0) { // while m is even.
    c++;
    m >>= 1;
}
return power2[c]^rank;
}

which you may use if you like. For pretty printing purposes I also wrote this method (which I know leaks memory like a rusty bucket!):

char *byte2bin(unsigned int x, int dim) {
    char* b = (char*) malloc(sizeof(char)*(dim));
b[0] = '\0';

    int z;
    for (z = power2[dim-1]; z > 0; z >>= 1) {
        strcat(b, ((x & z) == z) ? "1" : "0");
    }
    return b;
}

which turns a number into its binary representation as a string.

1.2 Implementation

I have designed a message type consisting of 25 integers (We could make it more complicated, but we do not really need that for this assignment).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
</table>

The content of a message is interpreted as follows:

<table>
<thead>
<tr>
<th>Position</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Source</td>
</tr>
<tr>
<td>1</td>
<td>Destination</td>
</tr>
<tr>
<td>2</td>
<td>Message Number</td>
</tr>
<tr>
<td>3</td>
<td>Type</td>
</tr>
<tr>
<td>4</td>
<td>Number of hops</td>
</tr>
<tr>
<td>5</td>
<td>First hop</td>
</tr>
<tr>
<td>6</td>
<td>Second hop</td>
</tr>
<tr>
<td>7</td>
<td>Third hop</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
In general, if $M$ is a message and $M[4] = n$ it means that the message went through $n$ hops (not including the source and destination) on its route in the hyper-cube. Furthermore, $M[5], \ldots, M[5+n]$ is the list of ids of the nodes the message went through to get to the destination. These ids will be added by the nodes as they are routed through the intermediate nodes. The source and the destination do not need to be added as we have those at the beginning on the message anyway.

For now, we only have one type of messages, namely MSG, so all messages are sent with this type. Let us start out with a program that looks like this:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#define SOURCE 0
#define DEST 1
#define MSGNO 2
#define TYPE 3
#define HOPCNT 4

#define MSG 0
#define MSGSIZE 25

A message can thus be defined as `int(MSGSIZE)`.

### 1.2.1 Sending Messages

Each node should send the same number of messages, namely the number passed in through the command line. The destination should be picked at random (it is ok to send a message to yourself). I have written a small function to make messages:

```c
int msgCounter = 0;

void makeMessage(int *msg, int source, int dest, int type, int msgno) {
    msg[SOURCE] = source;
    msg[DEST] = dest;
    msg[TYPE] = type;
    msg[HOPCNT] = 0;
    if (msgno > 0)
        msg[MSGNO] = msgno;
    else {
        msg[MSGNO] = source*100+msgCounter;
        msgCounter++;
```
For now, when calling this function pass in 0 for the msgno.

1.3 Receiving Messages

Two different kinds (not types) of messages can arrive at a node: messages destined for that node and messages that must be routed to another node according to the rules described in the previous section. If a message arrives that is destined for the node, it should be printed out. Here is an example (in a 3-dimensional hyper-cube):

4(100): Message (MSG) #201 from 2(010) to 4(100) in 1 hops (010->000->100)

which represents a message with number 201 sent from node 010 to 100 (which is node 4), and to get there it went through node 000.

1.4 Combining Sending and Receiving

We are going to impose the following rule: If there are incoming messages waiting to be dealt with, they must be taken care of first, and if there are no messages waiting, then we send one and go back and check if any messages have arrived.

This sounds easy, but it really is not. And before you get any funny ideas of writing multi-threaded code: DON'T!

There is an MPI primitive that can probe the runtime system for whether a message is available. It is called MPI_Probe(); however, this is a blocking call, and if the runtime system does not have any messages to deliver, MPI_Probe() blocks, so if every node calls MPI_Probe() as the first thing they do (which they must according to the rule above), everyone will block and the system is deadlocked. Luckily, another function MPI_Iprobe() is non-blocking, and returns immediately. Here is the man-page for it:

NAME
MPI_Iprobe - Nonblocking test for a message.

SYNTAX
C Syntax
#include <mpi.h>
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)

INPUT PARAMETERS
source Source rank or MPI_ANY_SOURCE (integer).
tag Tag value or MPI_ANY_TAG (integer).

comm Communicator (handle).

OUTPUT PARAMETERS
flag Message-waiting flag (logical).
status Status object (status).

If we call this function as MPI_IProbe(MPI_ANY_SOURCE, MPI_ANY_TAG, &flag, &status), flag will be 1 if there is a message, and by using the MPI_SOURCE field of the status record, we can find the source id, and now issue a regular MPI_Recv() with this source id. In this assignment all tags should be 0. We can now extract the destination id from the message and examine it to see if it should be printed or routed. You should keep probing and receiving for as long as there are messages in the runtime system: once you have dealt with all of them, go on and send a message if you have not yet sent all messages.

**Question 1:** Implement the hyper-cube routing algorithm and test it. The master process should run with rank = size – 1, that is, be the last process, and for this question it should not do anything.

2 Termination

As you will probably see, when you run your program, the messages are printed out correctly (after hours of debugging, no doubt), and then it all hangs; your processes will all be busy probing and not sending.

How do we terminate this gracefully? It is not as easy as it sounds! Imagine what would happen if once you have sent all your messages you simply terminate? What if the node was still needed for routing? Just because the runtime system does not have any messages at a certain point does not mean that there are no messages somewhere in the system.

A first approach could be to send a message to the master process with rank size-1 that you are done, and then wait for a termination signal? What if every node finished sending their messages, notified the master, and got a message back to terminate? The code of the master would look something like this:

```c
#define MASTER (size-1)
...
if (rank == MASTER) { // Master
    int i=0;
    // Receive a DONE message from each worker
    for (i =0; i<size-1; i++)
        MPI_Recv(msg, MSGSIZE, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
```
Question 2: Explain why this method will not work? You could try to implement it and see what happens.

To implement it we add two new types of messages:

#define DONE 2
#define STOP 3

2.1 Safe Termination

Consider the following scenario: If each node knows that all its messages have reached their destinations, it would be safe to terminate. However, your first implementation does not have that knowledge: A node does not know that all its messages have been delivered, and it does not know anything about the state of the entire system.

An obvious solution is for the recipient to send back a receipt (an acknowledgment) that the message has been received, and once a node has received as many acks (short for acknowledgments) as it has sent messages, it knows that all its messages have been received. If we then apply the technique of sending a DONE message to the master and wait to receive a STOP message, it should be safe to terminate. To accomplish this, yet another message type is added:

#define ACK 1

When a node received a message destined for it, an ACK message is sent back. The ACK message should be sent back with the same message number as the original message (pass it into the makeMessage method where you pass 0 for MSG messages.)

Here are two lines of output from my program with ACK messages added (this is the message and corresponding ack for message number 201):

4(100): Message (MSG) #201 from 2(010) to 4(100) in 1 hops (010->000->100)
... 2(010): Message (ACK) #201 from 4(100) to 2(010) in 1 hops (100->110->010)

Note, the ACK message did not take the same route as the MSG message.

Question 3: Implement the acknowledgment messages and make sure it works.
Question 4: Argue why the ACK packages and the rendezvous with the master is enough to ensure safe termination. In other words, explain why, when a node received a STOP message, no other message could be behind it, waiting to be routed, in the runtime system.

Question 5: Consider a small change to the system above. Instead of the master sending the STOP message to every node, and every node receiving a STOP message, could we replace the sending and receiving of the STOP message by a full barrier sync (MPI_Barrier()) between the master and the nodes instead?

3 Centralized Printing

On a real hyper-cube, it might not be possible to perform output from any of the worker nodes; if this is the case, all output must be sent to the master for printing. If we assume that only node 000...00 is connected to the master process, all other nodes must send their messages to node 000...00, which will forward it to the master.

If we want to implement such a system we can reuse the messages as they are without making any changes (almost). Instead of printing a message, the message type is changed to a new type:

```
#define PRINT 4
```

and the message is sent to node 00...0 (also remember to update the source field) which will then send it to the master, who will print it.

Naturally, we have now lost information about the original source (the original destination is still available as the source of the PRINT message). The fix for this is of course to alter the way the list of hops is maintained to include the source as the first hop. In addition, a new field needs to be added to the message so that a PRINT message know what type the message was originally. It can be added immediately after the TYPE field and everything else pushed down one.

To route a PRINT message, a node should not add itself to the hop list, just simply route the message.

Question 6: Make the changes necessary and implement the centralized printing as described above.

Question 7: Does the termination algorithm still work as before? Explain!

4 Non Loquuntur ad Dominum

Of course in the previous description we cheated a little when we implemented direct communication between the master and the nodes. That is something we
would not have in a hyper-cube; only node 0 can talk to the master.

Now, if you try to implement your termination algorithm by sending all the STOP messages to the individual nodes though node 000...00 in order 0,1,2,3,... things will go horribly wrong.

However, there seems to be a very simple solution to this problem, which does not involve computing spanning trees etc: Send the STOP messages in reverse through node 000...00. So first send a STOP message to node 111...11 then to 111...10 etc.

Question 8: Implement this new termination algorithm (Remember, no more communication with the master process unless your are node 000...00, so DONE messages must be routed as well. Also prove/argue for why this approach never kills off any nodes that need to be used for routing subsequent STOP messages.

5 Efficient Broadcast

In a hyper-cube all links can communicate at the same time, so we should be able to implement a very efficient broadcast protocol.

The basic idea is to build a spanning tree with node 111...11 as the root node; all broadcast messages should be sent to 111...11 and it initiates the broadcast utilizing the spanning tree. So let us look at how to compute a spanning tree in a hyper-cube.

5.1 Hyper-cube Spanning Tree

Each node in the cube must know who its children in the spanning tree is. We know that no node can have more than \( d \) (where \( d \) is the dimension) children, so we can keep the children in an array of length \( d \) and keep a counter of how many we have.

Although Lucas posted a better algorithm on the bulletin board (which requires exactly 0 communication), I’d like you to consider the following as an exercise in implementing parallel algorithms; after all that is what this class is about. The tree is constructed from the leaves and up with the following algorithm:

1. The master node send a TREE message to every node.

2. If a node receives a TREE message, it picks its parent in the tree. For a node \( i \), its parent is the neighbor node with the highest rank \( j \), where \( j > i \). If no such \( j \) exists, the node is the root of the tree (this is of course only true for node 111...11).

3. When the parent has been found, send a message of type PARENT to the parent.

4. If a node receives a PARENT message, it records the source id in its list of children. For every PARENT message a node receives a new PARENTFOUND message is sent to 111...11.
5. Once the counter in node 111...11 reaches $2^d - (d + 1)$ (111...11 does not have a parent so no PARENT message arises from that, also no PARENT messages arise from the $d$ immediate children of 111...11), the tree has been constructed, and a TREEDONE message can be sent to the master.

**Question 9:** Implement the Spanning Tree construction algorithm\(^1\). Add the following new message types:

```c
#define TREE 5
#define TREEDONE 6
#define PARENT 7
#define PARENTFOUND 8
```

It is possible to improve the process of building the spanning tree, by reducing the number of messages. This can be done in the following manner:

1. Master sends a TREE message to all nodes starting with a 0, i.e. 0XX...XX.
2. Send a PARENT message to your parent if you get a TREE message.
3. If you get a PARENT message, add the source of the message to your list of children if it is not already there. Then find your own parent if you have not already done so and send a new PARENT message to your parent.
4. When 111...11 has received $2^d - 1$ PARENT messages the tree is complete, and a TREEDONE message can be returned to the master.

**Question 9A:** Implement the improved Spanning Tree construction algorithm, and add the following message types:

```c
#define TREE 5
#define TREEDONE 6
#define PARENT 7
```

Now that we have a nice spanning tree, what can we do with it? We could use the tree for termination: have the master send a STOP message to 111...11; when a node receives a STOP message it sends it to all its children in the spanning tree and terminate. This would guarantee proper termination.

**Question 10:** Change your implementation of STOP so that the spanning tree is used to propagate STOP messages. The master should send the STOP message to node 111...11 though node 000...00 as usual.

\(^1\)See Question 9A, if you want you may implement 9A instead of 9.
6 Broadcast

The spanning tree is a convenient way of implementing broadcast: Send the message that you want to broadcast to 111...11 and start the broadcast from there. Each node, when receiving a message that is to be broadcast, forwards it to its children.

We need to introduce two new message types for this implementation:

```c
#define FORBCAST 9
#define BCAST 10
```

Use the `FORBCAST` when you want to broadcast a message; `FORBCAST` messages should always be addressed to 111...11. 111...11 creates a `BCAST` message and sends it to its children, who also forward the message to their children.

**Question 11:** Implement the broadcast algorithm, and test it, by having node 000...00 broadcast its last message rather than sending it to someone specific.

It might have occurred to you that we have a problem with termination again. The implementation of the broadcast algorithm can potentially pose a problem to the termination criterion: We cannot allow 111...11 to send a `DONE` message until we know that all the broadcasts have finished.

An easy way to accomplish this is to create two new acknowledgment message types:

```c
#define FORBCASTACK 11
#define BCASTACK 12
```

If a process has requested a broadcast and thus sent a `FORBCAST` message to 111...11, no `DONE` message may be sent until you have received a `FORBCASTACK`. This can be implemented with a simple counter which is incremented upon sending a `FORBCAST` message, and decreased upon the receipt of a `FORBCASTACK` message. Only when this counter is 0 may a `DONE` message be sent. The approach to acks for `BCAST` messages is similar, except, if you send/forward a `BCAST` message, you must increment your counter by the number of children you have, and only when you receive that many acks may you send an ack to your parent.

**Question 12:** Implement the broadcast acks in your system.

It should be noted, that this approach to broadcast only works if we only ever have one broadcast going on at a given time. This is because we do not distinguish between ACKs. If we want to allow multiple broadcasts to take place...
at the same time (though still only one from each process), we can change the
counter to an array of counters; if we want to allow multiple broadcasts (with
multiple from each node at the same time, we could use a hash table and the
message’s number as the hash).

7 Hamiltonian Cycles

Remember, a Hamiltonian in a cycle in an undirected graph is a cycle that
visits every node exactly once. If each node in a hyper-cube knows its next
and previous node in the Hamiltonian cycle, we can send a message of type
REDUCEFORWARD or REDUCEBACKWARD which will cycle either forward or back-
ward in the cube and visit every node and return back to the sender. This
allows us easily to implement reduce-computations across the entire node in an
efficient manner.

In order to compute the previous and the next node in the Hamiltonian cycle
in the cube, we need to compute the previous and the next values of the Grey
code of which the node’s binary id (address) is a member.

Here are the rules for computing the next hop’s address:

- If the node has address 100...00 the next hop’s address is 000...00.
- Otherwise, if the address has an even number of 1-bits, simply flip the
  least significant bit. If the address has an odd number of 1-bits, flip the
  bit to the left of the first 1-bit searching the bit-string from right to left.

To count the number of bits set to 1 (the parity) we can use this function:

```c
int parity(int n) {
    int c = 0;
    while (n > 0) {
        if (n%2 == 1)
            c++;
        n >>= 1;
    }
    return c;
}
```

To find the first position, searching from the right, that has a 1 in it, we can
use this function:

```c
int findFirst(int i) {
    int c = 0;
    while (i % 2 == 0) {
        c++;
        i >>= 1;
    }
    return c;
}
```
Remember, to flip a bit, we can use the power2 array to get the correct pattern with which we can perform an exclusive or operation.

**Question 13:** Implement a function `int myNext(int i)` which computes the next node in a forward Hamiltonian.

We can compute the previous node in the almost exact way:

- If the node has address 000...00 the previous hop’s address is 100...00.
- Otherwise, if the address has an odd number of 1-bits, simply flip the least significant bit. If the address has an even number of 1-bits, flip the bit to the left of the first 1 searching the bit-string from the right. (This is the same rule as we used to compute the next hop.)

**Question 14:** Implement a function `myPrev(int i, int dim)` which computes the next node in a backward Hamiltonian, i.e., compute the previous node in a forward Hamiltonian.

We need to add the two new message types:

```c
#define REDUCEFORWARD 13
#define REDUCEBACKWARD 14
```

We also need to add a new field to the message because we need to remember the source of the message; let us call it ORGSOURCE.

```c
#define SOURCE 0
#define DEST 1
#define MSGNO 2
#define TYPE 3
#define ORGTYPE 4
#define HOPCNT 5
#define ORGSOURCE 6
```

To use a REDUCEFORWARD the message’s ORGSOURCE is set to the id of the process that initiates the process and the message is sent to the next node in the Hamiltonian. When a process receives a REDUCEFORWARD a process must check if it were the source of the message, in which we are done. If we are not the source we can do something with the data of the message (if we had some!), build a new message with the same ORGSOURCE and forward it to the next node in the Hamiltonian. The REDUCEBACKWARD works the same way, just utilizing the previous node rather than the next. A REDUCE* message should retain it’s
original message number, and of course, no DONE message may be sent by the originating node for as long as there are outstanding REDUCE* messages.

**Question 15:** Implement and test the REDUCE* pattern described above.

**Question 16:** Could termination be implemented using a REDUCEFORWARD?

The REDUCE* operations are good (necessary) if your reduce operation is not commutative, but often reduce operations are. As an example, plus (+) is commutative, so the order in which the elements get added matters not. If that is the case we can speed up the reduction process from \( O(p) \) where \( p = 2^d \) to \( O(\log_2(p)) \).

The process of reducing parts of the result at different nodes parallelizes the reduction process. Consider the 3-dimensional hyper-cube, and let us as an example say that we want to compute the sum of a certain variable \( v \) with \( v_i \) being the instance in process \( i \) using this new reduction technique. All the nodes with odd ranks (i.e., with binary representations which end in 1) send their value to the neighbor obtained by flipping the last bit to 0. If you are an even numbered rank then you must behave slightly differently: The number of 0 bits (to the right of the first 1) determine how many messages you should receive before you can forward your part of the answer to the neighbor you obtain by flipping the right-most 1. Eventually, the result will end up at 000...00.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Initially</th>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( v_0 )</td>
<td>( v_0 \odot v_1 )</td>
<td>( v_0 \odot v_1 \odot v_2 \odot v_3 )</td>
<td>( v_0 \odot v_1 \odot \ldots \odot v_6 \odot v_7 )</td>
</tr>
<tr>
<td>1</td>
<td>( v_1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( v_2 )</td>
<td>( v_2 \odot v_3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( v_3 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( v_4 )</td>
<td>( v_4 \odot v_5 )</td>
<td>( v_4 \odot v_5 \odot v_6 \odot v_7 )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( v_5 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( v_6 )</td>
<td>( v_6 \odot v_7 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( v_7 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fast Reduce for a Commutative Operator \( \odot \).

We need a new type of message (again):

```c
#define FASTREDUCE 15
```

The way I described it, all such reductions will end up at 0000...00, however, unlike the previous reduction implementations, which could be initiated by everyone by simply sending a message to either the next or the previous hop in the Hamiltonian, the fast reduce operation has to be communicated to all nodes with an odd rank.
**Question 17:** Devise a way to implement the fast reduce; you might need to add additional fields to the message. Also now implement and test it. For example, you could use msg[24] as a data field for such tests.

And finally:

**Question 18:** To finish off, the tree construction should happen before we do any message sending. This is easy, just have the master send a new `INITDONE` message to everyone when the tree is constructed. Noone can send any messages before having received an `INITDONE`:

```
#define INITDONE 16
```

I am out of ideas to pester you with, so just one more question:

**Question 19:** How much time did you spend on this assignment, and what was the hardest part?
A Example Output

Here is the output of my solution to question 3 (Yeah, I did actually implement it and got it to work ;-) ), and I added a little statistics at the bottom just to see how many messages each node received:

```
mpirun -np 9 ./hypercube 3 3
```

1(001): Message (MSG) #100 from 1(001) to 1(001) in 0 hops (001->001)
1(001): Message (ACK) #100 from 1(001) to 1(001) in 0 hops (001->001)
7(111): Message (MSG) #701 from 7(111) to 7(111) in 0 hops (111->111)
7(111): Message (ACK) #701 from 7(111) to 7(111) in 0 hops (111->111)
7(111): Message (MSG) #600 from 6(110) to 7(111) in 0 hops (110->111)
7(111): Message (MSG) #301 from 3(011) to 7(111) in 0 hops (011->111)
5(101): Message (MSG) #101 from 1(001) to 5(101) in 0 hops (001->101)
5(101): Message (MSG) #200 from 2(010) to 5(101) in 2 hops (010->011->001->101)
5(101): Message (ACK) #502 from 1(001) to 5(101) in 0 hops (001->101)
5(101): Message (ACK) #500 from 3(011) to 5(101) in 1 hops (011->001->101)
1(001): Message (MSG) #102 from 1(001) to 1(001) in 0 hops (001->001)
1(001): Message (ACK) #102 from 1(001) to 1(001) in 0 hops (001->001)
1(001): Message (MSG) #502 from 5(101) to 1(001) in 0 hops (101->001)
1(001): Message (MSG) #702 from 7(111) to 1(001) in 1 hops (111->101->001)
1(001): Message (MSG) #202 from 2(010) to 1(001) in 1 hops (010->001->101)
1(001): Message (ACK) #101 from 5(101) to 1(001) in 0 hops (101->001)
3(011): Message (MSG) #500 from 3(011) to 5(101) in 1 hops (011->001->101)
3(011): Message (MSG) #301 from 7(111) to 3(011) in 0 hops (111->011)
5(101): Message (MSG) #200 from 2(010) to 5(101) in 2 hops (010->011->001->101)
5(101): Message (ACK) #502 from 1(001) to 5(101) in 0 hops (001->101)
5(101): Message (ACK) #500 from 3(011) to 5(101) in 1 hops (011->001->101)
1(001): Message (MSG) #102 from 1(001) to 1(001) in 0 hops (001->001)
1(001): Message (ACK) #102 from 1(001) to 1(001) in 0 hops (001->001)
1(001): Message (MSG) #502 from 5(101) to 1(001) in 0 hops (101->001)
1(001): Message (MSG) #702 from 7(111) to 1(001) in 1 hops (111->101->001)
1(001): Message (MSG) #202 from 2(010) to 1(001) in 1 hops (010->001->101)
1(001): Message (ACK) #101 from 5(101) to 1(001) in 0 hops (101->001)
3(011): Message (MSG) #500 from 3(011) to 5(101) in 1 hops (011->001->101)
3(011): Message (MSG) #301 from 7(111) to 3(011) in 0 hops (111->011)
7(111): Message (MSG) #402 from 4(100) to 7(111) in 1 hops (100->010->111)
7(111): Message (MSG) #602 from 6(110) to 7(111) in 0 hops (110->111)
7(111): Message (MSG) #0 from 0(000) to 7(111) in 2 hops (000->001->011->111)
7(111): Message (ACK) #702 from 1(001) to 7(111) in 1 hops (001->011->111)
2(010): Message (MSG) #601 from 6(110) to 2(010) in 0 hops (110->010)
2(010): Message (MSG) #2 from 0(000) to 2(010) in 0 hops (000->010)
2(010): Message (ACK) #202 from 1(001) to 2(010) in 1 hops (001->000->010)
3(011): Message (MSG) #501 from 5(101) to 3(011) in 1 hops (101->111->011)
3(011): Message (MSG) #400 from 4(100) to 3(011) in 2 hops (100->010->111->011)
3(011): Message (ACK) #300 from 6(110) to 3(011) in 1 hops (110->111->011)
4(100): Message (MSG) #1 from 0(000) to 4(100) in 0 hops (000->100)
4(100): Message (MSG) #700 from 7(111) to 4(100) in 1 hops (111->110->100)
4(100): Message (ACK) #401 from 6(110) to 4(100) in 0 hops (110->100)
4(100): Message (MSG) #201 from 2(010) to 4(100) in 1 hops (010->000->100)
6(110): Message (ACK) #600 from 7(111) to 6(110) in 0 hops (111->110)
6(110): Message (MSG) #300 from 3(011) to 6(110) in 1 hops (011->010->110)
6(110): Message (MSG) #401 from 4(100) to 6(110) in 0 hops (100->110)
7(111): Message (ACK) #700 from 4(100) to 7(111) in 1 hops (100->101->111)
5(101): Message (ACK) #501 from 3(011) to 5(101) in 1 hops (011->001->101)
6(110): Message (ACK) #601 from 2(010) to 6(110) in 0 hops (010->110)
6(110): Message (ACK) #602 from 7(111) to 6(110) in 0 hops (111->110)
```
2(010): Message (ACK) #200 from 5(101) to 2(010) in 2 hops (101->100->110->010)
2(010): Message (ACK) #201 from 4(100) to 2(010) in 1 hops (100->110->010)
0(000): Message (ACK) #2 from 2(010) to 0(000) in 0 hops (010->000)
0(000): Message (ACK) #1 from 4(100) to 0(000) in 0 hops (100->000)
4(100): Message (ACK) #402 from 7(111) to 4(100) in 1 hops (111->110->100)
4(100): Message (MSG) #302 from 3(011) to 4(100) in 2 hops (011->010->000->100)
4(100): Message (ACK) #400 from 3(011) to 4(100) in 2 hops (011->010->000->100)
3(011): Message (ACK) #302 from 4(100) to 3(011) in 2 hops (100->101->111->011)
0(000): Message (ACK) #703 from 7(111) to 0(000) in 2 hops (111->110->100->000)

Process 0(000): Sent 3 messages and received 0 messages and routed 4 messages.
Process 1(001): Sent 3 messages and received 5 messages and routed 4 messages.
Process 3(011): Sent 3 messages and received 3 messages and routed 4 messages.
Process 2(010): Sent 3 messages and received 2 messages and routed 3 messages.
Process 4(100): Sent 3 messages and received 4 messages and routed 2 messages.
Process 5(101): Sent 3 messages and received 2 messages and routed 5 messages.
Process 6(110): Sent 3 messages and received 2 messages and routed 5 messages.
Process 7(111): Sent 3 messages and received 6 messages and routed 5 messages.

Here is some code that works well for generating random numbers. If you want random numbers that are repeated for each run, then use rank instead of t.tv_usec as a seed to srand.

#include <sys/time.h>

... 

... 
struct timeval t;
gettimeofday(&t,NULL);
srand(t.tv_usec);
...
int dest = rand()%(size-1);
...