Parallel Computer Architecture PS1
Dinesh Reddy Chinta: dc47444
Tejas Bhagwat: tb34454

Problem 1 (10%)

Give three examples of inherently sequential, unparallelizable tasks. Give three examples of parallel tasks. Which was easier to come up with? *Solution should not exceed 1 page.*

Inherently sequential tasks:
Tasks that depend on the previous sub-task/previous step of the task can be considered inherently sequential or unparallelizable tasks.

Example 1 - Assembly Line:
Consider an assembly line in a factory that packages soda bottles before approving them for distribution.
Empty soda bottle -> Fill the soda in the bottle -> Package the bottle -> Check whether the packaged bottle has the required amount of soda/doesn't have any defects -> Send the bottle for distribution
Since each step of this process depends on the previous step, it's not possible to do 2 or more of these sub-processes in parallel. Hence, it's an inherently sequential, unparallelizable task.

Example 2 - Newton-Raphson's Method:
Consider an iterative process in Numerical methods such as Newton-Raphson's method. Since each step of the computation depends on the previous step, it's not possible to do these computations in parallel.

Example 3 - Relay Race:
Runners need to wait for the baton to be passed to them before starting to run. It doesn't make sense for the runners to run in parallel, given the format of the race.

Parallel Tasks:

Example 1 - Quiz Grading:
Grading a quiz of 50 students can be done in parallel by multiple graders.

Example 2 - Unrollable Loops:
Let's say that there is a for loop which assigns elements of an array

```
for(int i = 0; i < n; i++) {
    array[i] = f(i); // f is some function
}
```

Since there is no dependency between different iterations, the for loop can be "unrolled," and these operations can be done in parallel.

Example 3 - Parallalizable applications using SIMD:
The same operation can be done with multiple data elements in parallel. For example, adding a constant to each element in an array.

It was easier to come up with parallel tasks over strictly sequential tasks because we're trying to do increasingly more tasks in parallel to save time and increase efficiency.

Problem 2 (10%)

Give three examples of programs/tasks (you can use real-world examples) that you would prefer to parallelize with shared memory alone versus message passing alone. Explain why. Give three examples of programs you would prefer to parallelize with message passing alone versus shared memory alone.

Shared memory over message passing:

Example 1 - Lecture Broadcasting:

Consider a professor giving a lecture in a class to students. (The class being “shared memory”). Every piece of information provided by the professor is “new data.” In this scenario, it’s better to have “shared memory” over “message passing.” Message passing would be the professor notifying all the students regarding the new information. A lot of time and resources would be wasted in such a scenario.
Hence, in this case, it’s better to have shared memory over message passing.

Example 2 - Producer-Consumer:

The consumer can consume data produced by a producer from a shared buffer. Since the producer and consumer updates the buffer frequently, the overhead of message passing would be very high, and hence it makes sense to have a shared memory buffer in this example.

Example 3 - Multiplayer Game:

Consider a multiplayer game with states (scores, player positions, etc.) for each player. In this case, it is easier to do it over shared memory since the states are constantly updated, and each player can view the changes. Each player can see an identical copy of the scores.

Example4

Shared memory could solve the problem of unreliable communication where the messages are lost due to the destination being inactive. Shared memory inherently solves this problem.

Message passing over shared memory:

Example 1 - Spatially Segregated Nodes:

Message passing is required when it’s challenging to implement shared memory. One of the cases might be when the processors are physically far from each other.
Consider multiple IoT edge processors operating on a network. In this case, message passing would be highly preferred over shared memory.

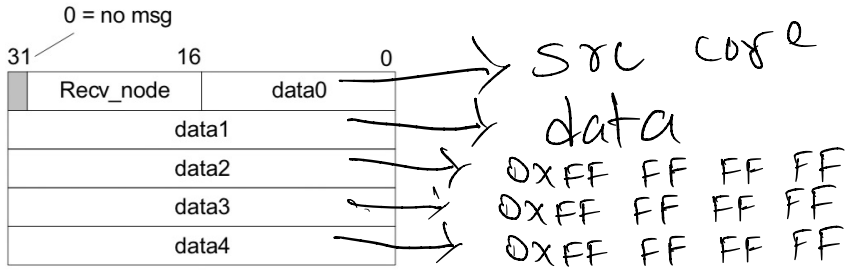
Example 2 - Phone Calls & Watsapp:

Shared memory does not scale well with the number of nodes. If a million nodes are communicating, then a shared memory implementation would need a designated address for every combination of node pairs; otherwise, shared memory should have one common buffer where every node searches through the entire buffer to get messages destined to it. Phone-Call or user messaging application like WhatsApp captures this concept.

Example 3 - Security Requirement:

A shared memory poses the danger of leaking information if the users gain access to the central shared memory node. Message passing system ensures data segregation by having the spatial barrier.

Problem 3



To start with, we are only concerned with the correctness and ease of the implementation and not the performance. Here, to ensure correctness, we need to send a 5-word message across the bus so that the receiving core correctly interprets the message. To simplify the implementation, we are embedding the data as annotated in the above picture.
1. The source core, used by the receiver while checking if the msg has arrived for the required core, is embedded in “data0”.
2. The data in the one-word message is embedded in the “data1” field of the message.
3. The other words in the message are simply wasted by writing all 1s (i.e., binary -1). We chose to send 1s instead of 0s to make sure that data is actually being sent.

Note that for each word the application sends, we are sending a 5-word message across the bus, which is very inefficient.

The below code snippets shows the implementation of such receiveMsg function.

```
int resendMsg(int dest, int data, int src)
{
   // check if the msg can be sent
   if(! *OutStatus) return 0;

   enqueue(OutFIFO, 0x80000000 | (dest << 16) | src);
   enqueue(OutFIFO, data);

   enqueue(OutFIFO, -1); enqueue(OutFIFO, -1); enqueue(OutFIFO, -1); // dummy words

   return 1; // return successful msg sent
}

//receives one word from the source node “src”)
int receiveMsg(int src)
{
       do{
      int word1 = dequeue(InFIFO);
      if (word1 & 0x80000000){ // check if the msg is valid

         // extract the rest of the message
         int word2 = dequeue(InFIFO);
         dequeue(InFIFO); dequeue(InFIFO); dequeue(InFIFO); // discard

         // check if the dst matches
         if((word1 & 0xFFFF) != src){
            return word2;
         }
         else{
            while(!resendMsg(word1 & 0xFFFF, word2, word1 & 0x7FFF)); // push to the back of
the InFifo
         }
      }
       } while(1);
}
```

Implementation details:

Once the application requests data from a specific core, we block on the InFIFO until we find the requested data; more specifically, we poll for the input by constantly reading the InFIFO until a valid message from the required core is found. We need to make sure that the messages sent by other cores that the application might request later on during the execution are not lost while we are waiting and unqueuing for the messages from a specific core inside receiveMsg. To solve this, we resend all the valid messages originating from a different core into the bus with the destination as the current core. Notice that we can't simply call sendMsg since that would change the source core to the current core, which would lose the actual sender information and hence be incorrect.

The below code snippet shows the implementation of SendMsg

```
#define CORE 1
int * InFIFO = (int *) 0x1234;
int * OutFIFO = (int *) 0x2345;
int * OutStatus = (int *) 0x3456;

// subroutine declaration for operating on InFIFO
and OutFIFO
void enqueue(int* f, int data);
int dequeue(int* f);

int sendMsg(int dest, int data)
{
   // check if the msg can be sent
   if(! *OutStatus) return 0;

   enqueue(OutFIFO, 0x80000000 | (dest << 16) |
CORE);
   enqueue(OutFIFO, data);

   enqueue(OutFIFO, -1); enqueue(OutFIFO, -1);
enqueue(OutFIFO, -1); // dummy words

   return 1; // return successful msg sent
}
```

The implementation of sendMsg is exactly the same as resendMsg except that we need to preserve the essential source information, which we pass as a parameter to the resendMsg function. In the function, we first check if the entire message can be sent by checking the status flag, as described in the question, by assuming the status flag represents whether the entire 5 words can be sent, not just one word. If the status check fails, we immediately return with 0; otherwise, we create the first two words corresponding to the SRC, DST address, and the actual data, respectively. Then we simply send -1 (0xFFFF_FFFF) for the remaining three words to ensure that the entire 5 messages should be sent for a message. We finally return 1 to indicate that the sending is successful.

Maximize performance. What is your measure of performance? Is it as general as possible? What are the issues?

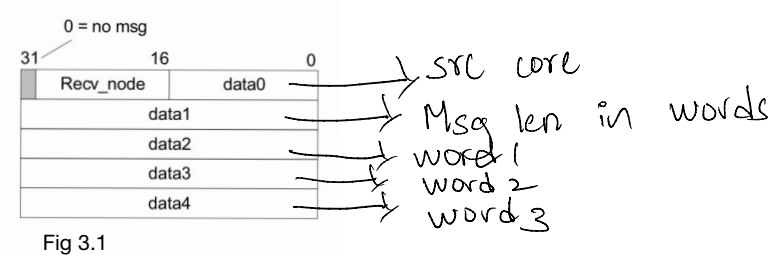
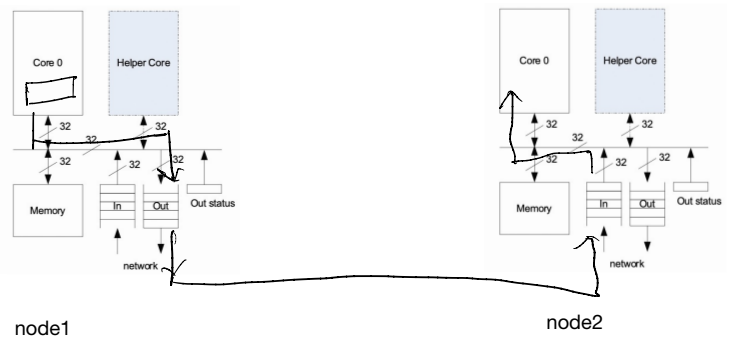


Fig 3.1

Latency and Baudrate are two general performance metrics relevant to our problem. Let us analyze our current implmentation interms of both the metrics.

1. Latency:
We define latency as the time it takes for a word to travel from the source core to the destination core. The data flow is as shown below:



node1 node2

It involves four major components:
1. time spent in the outFifo. A larger FIFO means more latency.
2. time spent in the network. We do not have control over this.
3. time spent in the inFIFO. Besides the length of the FIFO, reinsertion latency, because the application requests msg from a different core.

Out of the three, we can optimize the receive latency using software queues as a placeholder instead of reinserting them into the network. Consider the below implementation for the software fifos. As shown in Fig 3a. We have implemented a msg_bank data structure which is a 2-d linked list where, each node holds a linked list corresponding to a specific core (src attribute) in the attribute head and a pointer to next node. Each node in the core specific linked list pointed to by the “head” attribute stores a msg field pointing to the messages. We use the following intuitive functions to modify the msg_bank data structure.

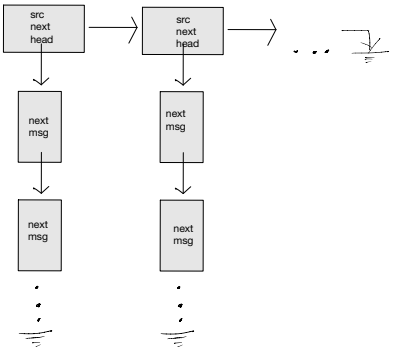


Fig 3a. struct msg_bank

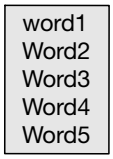


Fig 3b. struct msg

1. ll_node_t* search_msg_bank(int src);
   It is used to search msg_bank and remove the head element in the linked list while checking if there are some messages in the software buffers corresponding to an src.
2. void add_to_bank(msg_t m){
   Adds a message to the bank. This is called to store messages into the bank temporarily.

```
//receives one word from the source node "src")
int receiveMsg(int src) {
   // first check if a message exists in message_bank
   ll_node_t*ll = search_msg_bank(src);
   static msg_t msg;
   static int current_read = 0;

   if (current_read == GET_MSGLEN(msg)){ // need to read a new msg
      if(ll == NULL){ // no pre existing messages in the bank
         do {
            msg  = __recieve_one_msg(); // extract one msg from InFIFO
            add_to_bank(msg); // add it to the bank
         } while(GET_SRC(msg) !=  src); // search until a msg is rcvd

         ll = search_msg_bank(src); // remove the msg from the bank
         msg = *(ll->msg);
      }
      else{ // pre existing msg in the bank
         msg = *(ll->msg);
      }

      current_read = 0; // reset the # words read
   }

   int ret_val = (current_read == 0) ? msg.word3
               : (current_read == 1) ? msg.word4
               : msg.word5;

   current_read++;
   return ret_val;
}
```

2. Bandwidth: In our baseline implementation, it is not hard to see that for every word of data, we are sending 4 extra words, 1 for metadata, and 3 garbage data. To improve this, we can send multiple words at once by maintaining the msg length in one of the fields; In other words, in the best case, we can send 3 words at once, where we assign one word for communicating the message length. Fig 3.1 has the details of the new message structure. However, it is essential to discuss the implementation and identify the scenarios where this implementation works and where it doesn't.

From the receiver end, our previous implementation, which checks for the message length, also works for our current maximum bandwidth implementation.

For the SendMsg function, when the core streams data to another core, we can temporarily store 2 previous words in the function and write them into OutFIFO when a third word to the same core is sent. However, if the core sends a word to a different core, we stop waiting for more words in the previous message and send with whatever currently available words are. Furthermore, we maintain a timer to avoid waiting infinitely for a word to be sent towards the current destination node. The below code snippet shows the implementation of such SendMsg.

```
extern int* TIMER;

#define RESET_MSG(dest,data)\
   pre_dest = dest; word3 = data; SET_TIMER()

// microseconds
#define TIME_CONSTANT 1
#define SET_TIMER()\
   *TIMER = TIME_CONSTANT

static int word1, word3, word4, word5;
static int curr_num_words = 0, pre_dest = 0;

int _send_curr_msg() {
   // check if the msg can be sent
   if(! *OutStatus) return 0;

   word1 = 0x80000000 | (pre_dest << 16) | CORE;
   enqueue(OutFIFO, word1); enqueue(OutFIFO, curr_num_words);enqueue(OutFIFO, word3);
   enqueue(OutFIFO, word4); enqueue(OutFIFO, word5);

   curr_num_words = 0;
   return 1; // return successful msg sent
}

void timer_interrupt_handler(){
   _send_curr_msg();
}
```

```
int sendMsg(int dest, int data) {
   if((curr_num_words > 0) && (curr_num_words < 3)){ // previous words available
      if(pre_dest == dest){ // adding to previous words
         curr_num_words++;
         switch(curr_num_words){
            case 1: // save the word for later
               word3 = data; SET_TIMER();
               return 1;
            case 2: //save the word for later
               word4 = data; SET_TIMER();
               return 1;
            case 3: // send the msg
               word5 = data;
               return _send_curr_msg();

            default: // never taken
               return 1;
         }
      }
      else{ // curr dest different from prev msgs
         // send the previous message
         _send_curr_msg();
         RESET_MSG(dest,data);// form a new message now
         return 1;
      }
   }
   else{ // curr_num_words ==  0
      RESET_MSG(dest,data);// form a new message now
      return 1;
   }
}
```

Our current implementation takes full advantage of sending words to the same destination core. However, if the node interleaves between destination nodes for each word sent then our sendMsg's performance degrades to our baseline implementation.

A solution to such an interleaving scanario is instead of maintaining one set of previous words and timers for each destination core, we maintain a seperate set of previous word and timer for each destination, like we did for our receiveMsg with our msg_bank data structure. Standard Operating Systems allows applications to have multiple timers.

An even more simpler solution would be to write to the FIFO periodically in a timer. This would reduce the number of timer to just 1 and also the number of interrupts to be just 1.

If you are given Helper Core (sitting on the coherent shared memory bus with Core 0) as a dedicated processor for handling messages, can you further improve performance?

While the side core does not entirely improve the latency and bandwidth performance metrics we discussed above, it could be used to improve the CPU performance of our main core by avoiding waiting for a message in the receiveMsg function. Since the side core handles messages and writes to the msg_bank data structure, the main core's application could be enhanced to sleep and context switch to another runnable application. Indeed, several applications could start receiving messages through the msg_bank structure.

Problem4

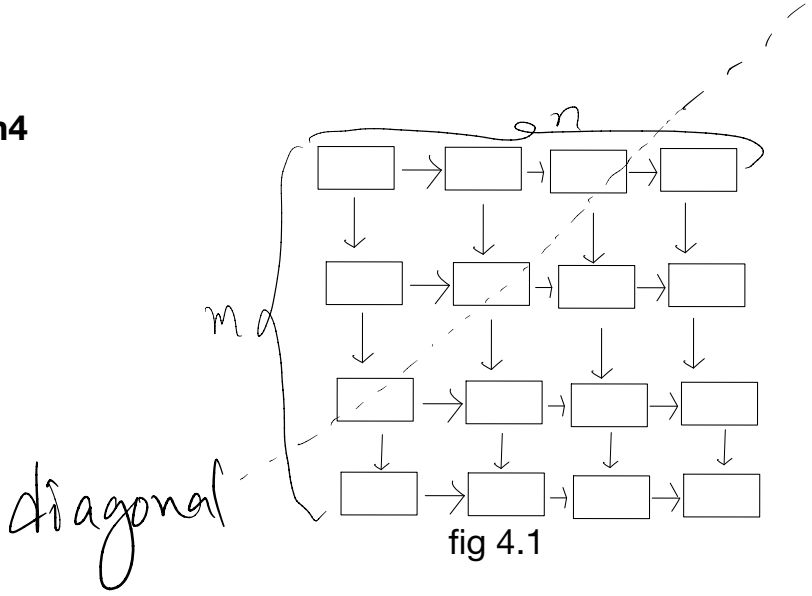


fig 4.1

Let's try to understand the problem and get a dataflow graph for the computation to identify the parallelizable and sequential portions of the computations. Given the transformation function in the question,

W[0, 0] = constant
W[x, 0] = f(W[x-1, 0], 0) if x > 0
W[0, y] = f(0, W[0, y – 1]) if y > 0
W[x, y] = f(W[x-1, y], W[x, y-1]) if ((x > 0) && (y > 0))

Using the equation, we can build the dataflow graph shown in Figure 4.1. We can note that every computation along the diagonal shown in the figure is independent of each other and thus can be parallelized. To exploit maximum parallelism, we have a separate processing element to carry out computation along the diagonal. Therefore, the number of processing elements in our system is

diag = min(m, n)

The number of steps it takes to complete the computation is,
steps = m + n -1

Implementation: We need to assign memory and computation to each of the cores and trigger the core correctly when the data is being communicated. We assign computation(i,j) to the core corresponding to the column number.

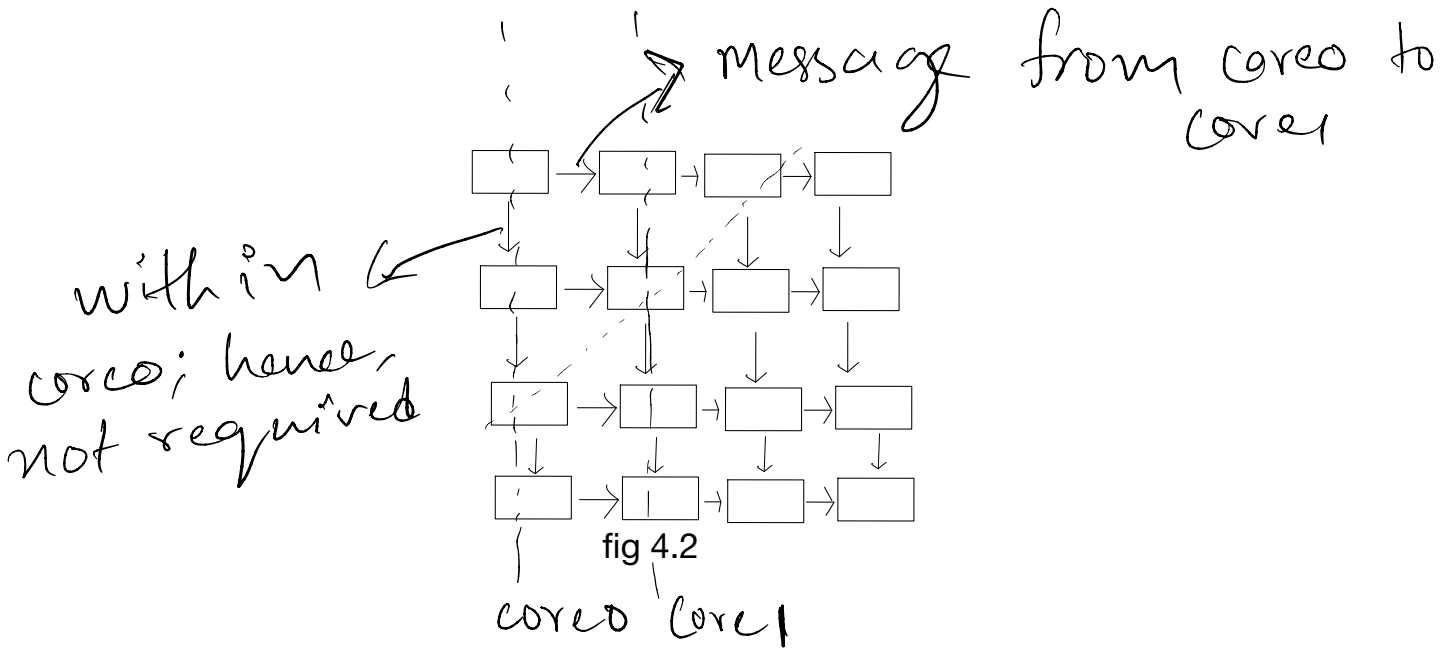
computation(i,j) => core_j

Let us implement this solution in message communication and shared memory systems.

1. Message Passing: we can note in Fig4.1 that every communication to the right should be sent as a message, and every communication to down need not be transmitted since it is the same core.

messages to be transmitted = (#rows) * (#columns -1) = m*(n-1)

If message communication is our main optimization goal, we could have run everything on a single core and hence end up with no messages at all. Since our primary goal is parallelism and the minimum no of messages is O(m*n).



Memory distribution: each computation result will be stored in the memory corresponding to the core in addition to being distributed.

```
#include "transform.h"

#define CONSTANT 100
#define N 10        // matrix dimensions

int result[N];        // memory to store the result and communicate

int main() {

    int left_val = 0;
    int row =0;
    int column =4 ;  // current row and column no of the core

    for(; row < N; row++){
        if((row == 0) && (column == 0)) {
            result[row] = CONSTANT;
        }
        else if(row == 0) {
            RECEIVE(&left_val, sizeof(int), row*N + column - 1);
            result[row] = transform_function(left_val, 0);
        }
        else if(column == 0){
            result[row] = transform_function(0, result[row-1]);
        }
        else{
            RECEIVE(&left_val, sizeof(int), row*N + column - 1);
            result[row] = transform_function(left_val, result[row-1]);
        }

        // communicate the result to right and bottom cores. We assume that last
        // row and column won't raise an error when sending data to a non-existat
core
        SEND(result + row, sizeof(int), row*N + column + 1);
    }

    return 0;
}
```

We assume that the SEND and receive messages are given to us. Here, since each core has only one computation to do, we do not need any complicated synchronization.

2. Shared memory

Our solution, where we first exploited parallelism, also applies to a shared memory system. However, in the implementation, we use mailboxes to communicate data between cores.

```
#include "transform.h"

#define CONSTANT 100
#define N 10

int mail_boxes[N*N];
int result_matrix[N*N];

inline void mail_check(int row, int column) {
    while(mail_boxes[row*N + column]);
}

int main() {

    int row =0;
    int column=4;
    int left_id = (row*N) + (column - 1), top_id = (row-1)*N + column;
    int core_id =  row * N + column;
    int right_id = row*N + column + 1, bottom_id = (row+1)*N + column;

    for(; row < N; row++){
        if((row == 0) && (column == 0)) {
            result_matrix[row*N + column] = CONSTANT;
        }
        else if(row == 0) {
            mail_check(left_id, column);
            result_matrix[row*N + column] = transform_function(result_matrix[left_id*N + column], 0);
        }
        else if(column == 0){
            mail_check(row, top_id);
            result_matrix[row*N + column] = transform_function(0, result_matrix[row*N + top_id]);
        }
        else{
            mail_check(left_id, top_id);
            result_matrix[row*N + column] = transform_function(result_matrix[left_id*N + column], result_matrix[row*N + top_id]);
        }

        mail_boxes[row*N + column] = 1;
    }

    return 0;
}
```

For the message passing code, indicate how the matrix will be stored. An equal number of elements must be stored in each processor's memory.

int result[N]; // memory to store the result and communicate