Parallel Programming Assignment 1

Shyam Kantesariya

ID#40042715

Question 1:

a) Serial version of the program:

Code file name: *serial.c*

Github

b) Parallel version of code with pre-defined number of processes

Code file name: *static_child.c*

Github

c)

Granularity of task: As granularity is determined by number and size of tasks into which a program can be decomposed, Number of DARTS being thrown by each task and total number of processes determines the granularity in our case.

As system is embarrassingly parallel, we can divide the work close to equally among all processors as long as #processors <= #DARTS.

Program category: Data parallel, and SPMD as we are basically **dividing the data** that is number of DARTS being thrown during each iteration keeping **same code** for all processes.

d) Parallel version of code with Dynamic child processes using MPI_Comm_Spawn

Master code file name: *dynamic_child_master.c*

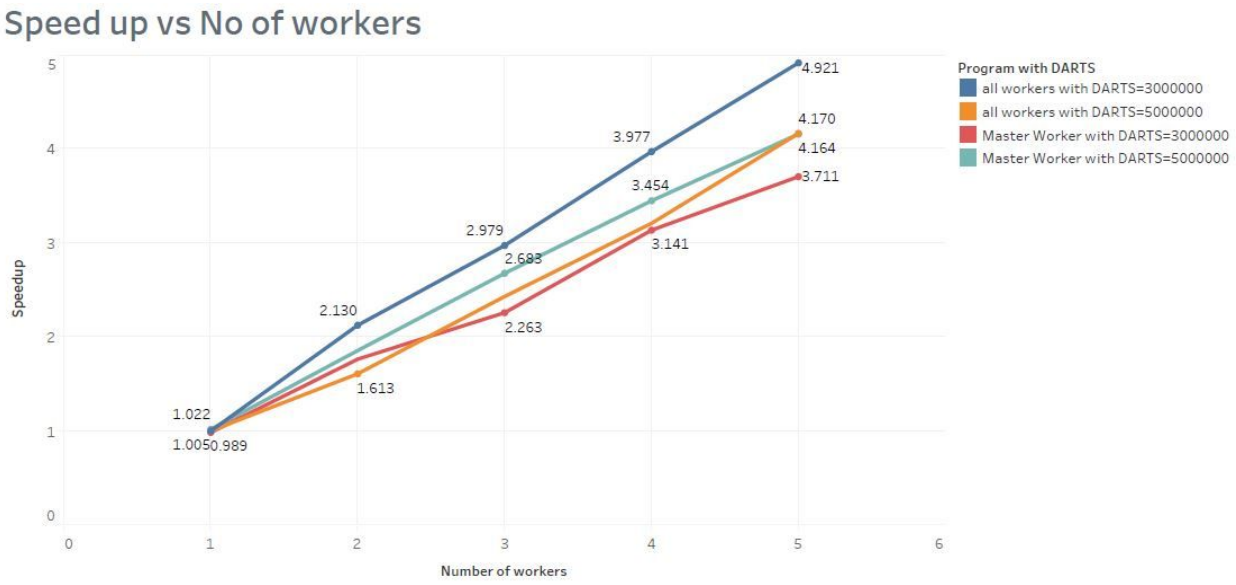
Github

Worker (child) code file name: *dynamic_child_worker.c*

Github

e) Speedup vs Number of workers

Following is the graph for Speed up vs No of workers for two different values of DARTS.



<Please find better visual image in PDF file "speed up vs no of workers.pdf" in package>

Superlinear Speedup:

We could see one considerable instance of superlinear speedup that is while running parallel program with all workers model, #DARTS=3000000 for each iteration, speedup is 2.13 while running with 2 workers.

Justification : It could be that for that particular time window cluster has the least load and working at its best. There should not be any other reason because If we consider this as usual behaviour then we should observe the same behaviour while running master-worker code with same number of workers and DARTS but that is not the case.

Sublinear Speedup:

We can generalize the system running around sublinear speedup as for majority of the cases, speedup is considerably below #workers assigned.

Justification: When we think of a computing system deployed on a cluster of machines, there is always a cost added for process communication. Thus, network latency plays a significant role in execution time.

Apart from that as we create new workers, there is an overhead to initialize a new process while launching a worker.

Most of the time these costs are overlapped if system is highly computing intensive but ours is not.

Hence, we see sublinear speedup while comparing parallel runtime with serial.

Question 2:

#================ MASTER CODE ====================#

```
DARTS = 100 //Initialize darts with some value which defines granularity of system
ITERATION = 0;
avepi = 0;

CALCULATE_PI():
        comm child_comm;
        parent_process_id = process ID of master process
        /*
         * create child processes to fill all the available slots while starting the program
         * This approach is good for this system as its embarrassingly parallel
         */
        CREATE_CHILD_PROCESSES(child_comm, GET_AVAIABLE_PROCESSORS());

        while (TRUE):
          /*
           * Get the result from child process along with its process id
           * Process id is required to assign a new task to that child process
           */
          (pi,process_id) = RECEIVE_RESULT_FROM_CHILD()

          if CHECK_IF_PI_CONVERGED(pi):
            /*
             * Good that we have got the converged value of PI so lets abort all pending child and finish
      the process
             */
            print "Value of PI is avepi"
            ABORT_ALL_CHILD_PROCESSES(child_comm)
```

```
            break;
          else:
            /*
             * We haven't reached till the converged value of PI so lets go for some more iterations
             */
            ASSIGN_NEW_TASK_TO_CHILD(process_id)
            CREATE_CHILD_PROCESSES(child_comm, GET_AVAIABLE_PROCESSORS(parent_process_id))

RECEIVE_RESULT_FROM_CHILD():
        wait for response from any child process;
        on receiving a response
        return (value, process_id)

GET_AVAIABLE_PROCESSORS(int process_id):
        //Based on resource allocation and utilization logic
        return #processors process process_id can use.

CREATE_CHILD_PROCESSES(comm child_comm, int n):
        if n <= 0:
          return
        else:
          spawn n new child processes within child_comm

ABORT_ALL_CHILD_PROCESSES(comm child_comm):
        /*
         * There could be two approaches to abort child process
         * 1. send abort message and let child process handle it to exit safely
         * 2. kill the child process forcefully
         * We have followed the former one
         */
        send abort message to all processes within child_comm
        return when all child processes have aborted

CHECK_IF_PI_CONVERGED(double pi):
        pi = value returned by child process;
        avepi = ((avepi * ITERATION) + pi)/(ITERATION + 1);
        ITERATION = ITERATION + 1;
        /*
         * assumption that if avepi is close to 3.14159 then its converged.
         */
        return mod(avepi - 3.14159) < 0.00001

ASSIGN_NEW_TASK_TO_CHILD(int process_id):
        send message to process process_id with new task to work upon

#================ WORKER CODE ====================#

WORKER:
```

```
        /*
         * When task is created assume that it has been assigned a task to work upon
         */
        bool new_task_assigned = TRUE;
        while (new_task_assigned):
          EXECUTE_TASK();
          new_task_assigned = MESSAGE_FROM_MASTER();

EXECUTE_TASK()
        /*
         * Execute the task and return local pi value to master
        throw DARTS number of darts;
        calculate pi value;
        return pi to Master;

MESSAGE_FROM_MASTER():
        /*
         * process message received from Master process
         */
        wait for message from master process;
        if received message is to execute new task:
          return TRUE;
        else:
          return FALSE;
```

Question 3:

a)

- Minimum number of processors required to obtain the lowest execution time is **10**
- Corresponding efficiency

  E = S/p = Serial Cost / Parallel Cost

  Serial Cost = 1x1 + 10x2 + 5x5 + 1x1 = 47

  Parallel Cost = 10 (1+2+5+1) = 90

  Hence, E = 47/90 = 0.52

b) Which one can increase the efficiency

1. Increasing the number of processors

No. As granularity of system is 10 so any number of more processors will increase overhead because of idling time.

2. Decreasing the number of processors

   Yes. It will reduce the time processors spend remain idle hence increase the efficiency.

3. Changing to faster processors

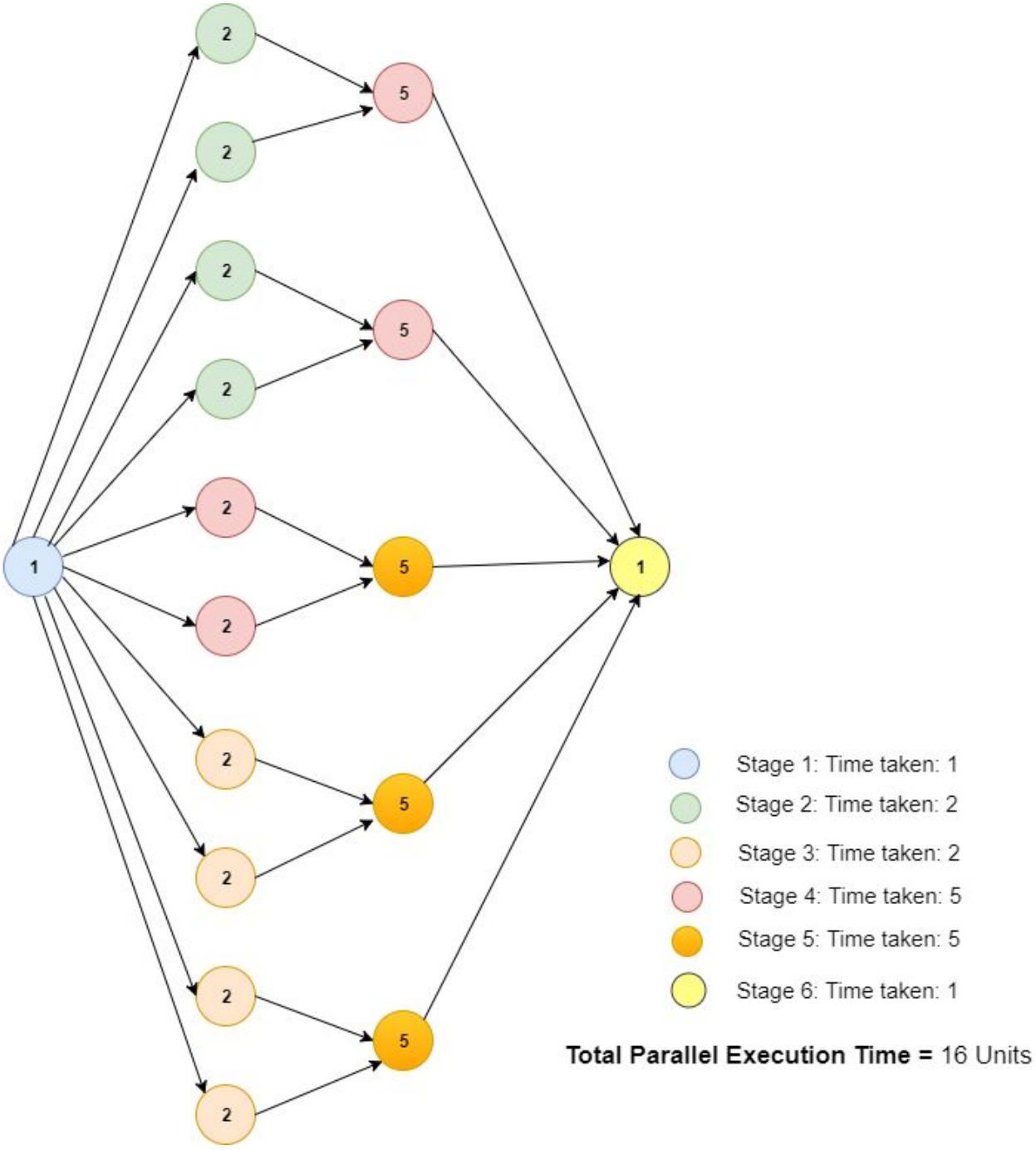
   Not really. The reason is if we make processors faster by some x factor than that will be counted in serial runtime as well, hence ratio of serial to parallel runtime will remain same in that case.

c) maximum speedup of the above parallel system when it is solved using 4 identical processors

Serial Runtime Ts = 47

Parallel Runtime Tp = 16

Following image explains the calculation of parallel runtime



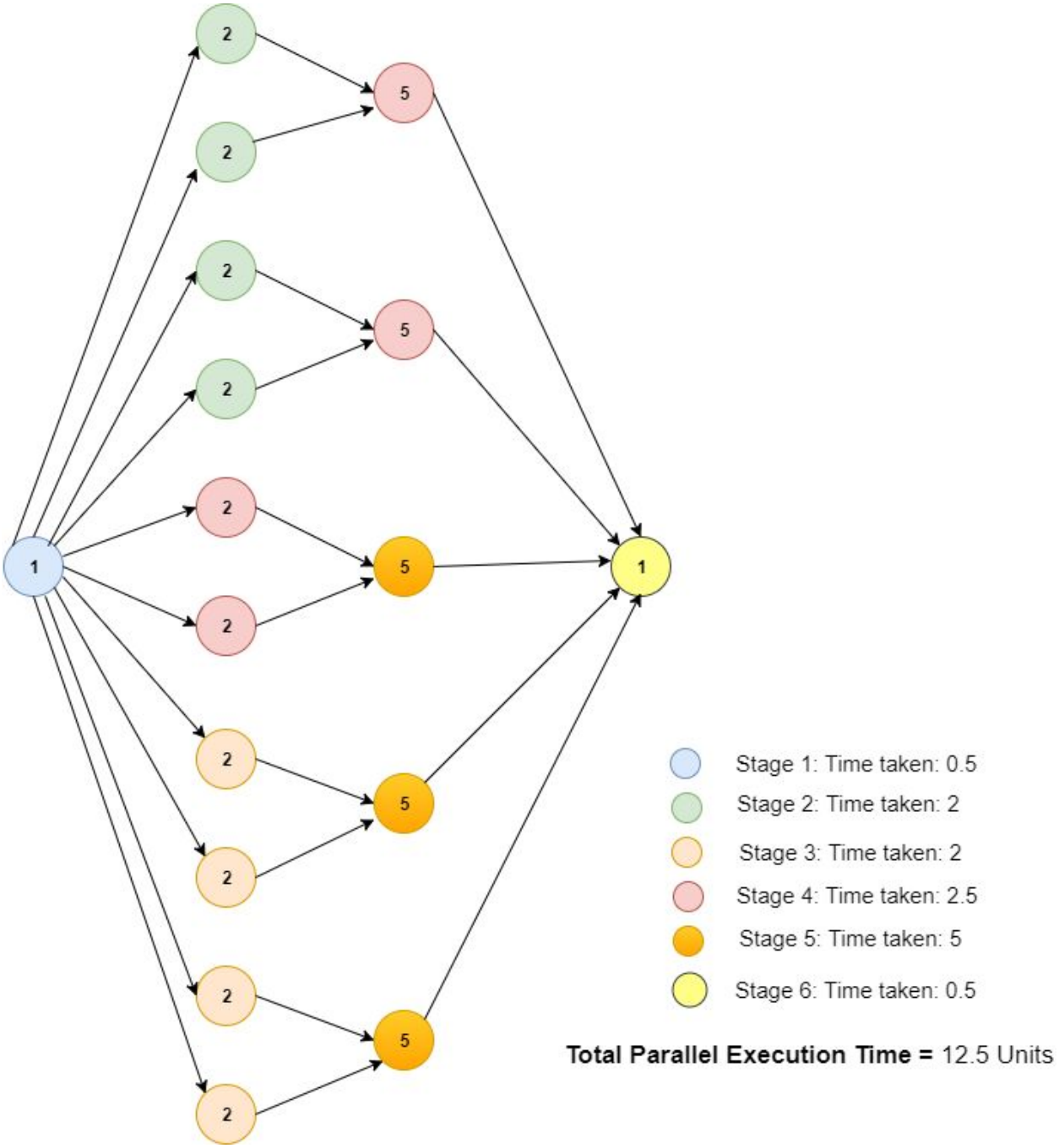
Hence Speed up S = Ts/Tp = 47/16 = 2.937

d) maximum speedup of the above parallel system when it is solved using 4 processors of varying speeds as follows: processors 1 and 2 have identical speed; processors 3 and 4 have identical speed; processor 1 (processor 2) is twice as fast as processor 3 (processor 4)

Serial Runtime Ts = 23.5 [Considering we run whole process on fastest processor

Parallel Runtime Tp = 12.5 [Assumption is to utilize faster processors first]

Following image explains the calculation of parallel runtime



Hence Speed up S = Ts/Tp = 23.5/12.5 = 1.88

Question 3:

a)

Consider we have N number of processors to sort N values.

Serial Runtime Ts = O(NLog(N))

Parallel Runtime Tp = O(N) as after every divide step, work will be divided by a factor of 2 compared to previous step.

Hence Tp = N + N/2 + N/4 + N/8 + N/26 + ... + N/Log(N) = 2N - 1 = O(N)

So Max Speed up S = Ts/Tp = NLog(N) / N = Log(N)

b)

As explain in (a)

Ts = O(NLog(N)) and Tp = O(N)

Hence Efficiency E = Ts/PxTp = **NLog(N) / NxN = Log(N)/N**

C)

No.

The reason is each partitioning step is serial, hence we have processors remain idle during at least

(Log(N) - 1) steps that contributes to the To (Overhead) and reduces the efficiency.

Question 5:

Since any path from a start to a finish can not be longer than l,there must be at least t/l independent paths from start to finish nodes to accommodate all t nodes. So d must be >t/l Then If d>t-l+1,then it is impossible to have a critical path of length l or higher because l-1 more nodes are needed to construct this path.So t/l <= d <= t-l+1

Even if we consider d tasks running at each level of critical path, dl = t

if dl > t that means we have more than d tasks running at some level along the critical path that is our statement “d is the degree of concurrency” is wrong.

We can understand the same with example as:

Case 1: l>1

t=10, d=5, l = 2:

(t/l=5) <= (d=5) <= (t-l+1 = 6)

Case 2: l = 1

t=10, d=10, l = 1 means all tasks run in parallel hence t=d

(t/l=10) <= (d=10) <= (t-l+1 = 10)

[**Source**: Solution of exercise given in reference book: Introduction to Parallel Computing, Second Edition

By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar]