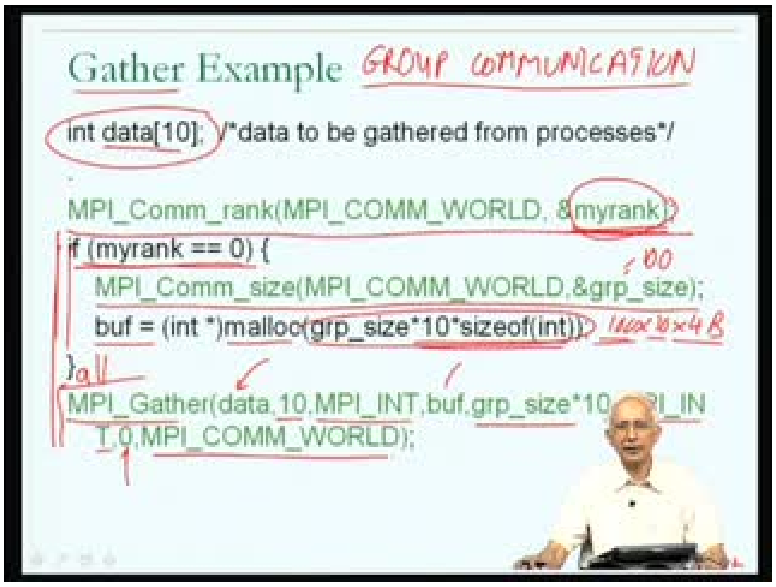
High Performance Computing
Prof. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 09
Lecture No. # 41

Welcome to lecture 41 of our course on high performance computing, this is the last lecture in this course. In the previous lecture, we had learned a lot about the different functions of the MPI Message Passing Library, which is what a lot of people use to write their parallel programs to run on parallel architectures or clusters of work stations using message passing as the mechanism for communication and interaction between processes.

(Refer Slide Time: 00:41)



Gather Example

```
int data[10]; /*data to be gathered from processes*/
.
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Comm_size(MPI_COMM_WORLD,&grp_size);
   buf = (int *)malloc(grp_size*10*sizeof(int));
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_IN
   T,0,MPI_COMM_WORLD);
```

We had looked at the different versions of send and receive, which are used for communication in MPI; some of the underlying very important concepts such as the communicator and the rank etcetera.

We were looking at an example of one particular kind of group communication which is provided by MPI. Group communication is the idea that, since there are frequently situations where we want one process to communicate with a group of processes, where might not be that efficient for the programmer to talk about individual sends and receives to between one process, and let say hundred other processes. MPI fortunately provides us with a single function which can be used for a process communicate with a group of processes.

And they could be hundred for thousand processes within that group; one such example is the gather group communication in function through which data can be collected from each of the individual processes, from the parallel program into one of them and that is what why it is known as a gather. So, in this particular example which I have just go through once again, we have a situation where there are some number of processes which are part of the parallel program.

Now, any MPI program will be run so that the same program, the same executable will be running as for each of the processes and each of the processes. Therefore it is, typically the case that each of the processes will check to find out what its own identity is and in the subsequent parts of the program, the program will be written, so that based on which process you are talking about it will do either one thing or another.

So, in this particular example after all the processes have determined who they are, in terms of what their rank is one particular process, which has the rank of 0 actually finds out what the size of the communicator is; in other words it tries to find out how many processes in some sense this program is running else.

So, if the process is running, if the program is running as the hundred processes then it will set its variable group size to 100. And what is subsequently going to be done is, it allocates a buffer so the process called zero, allocates a buffer within its virtual address space using malloc; of adequate size to allow for each of the hundred processes involve from the in that communicator to send it ten integers.

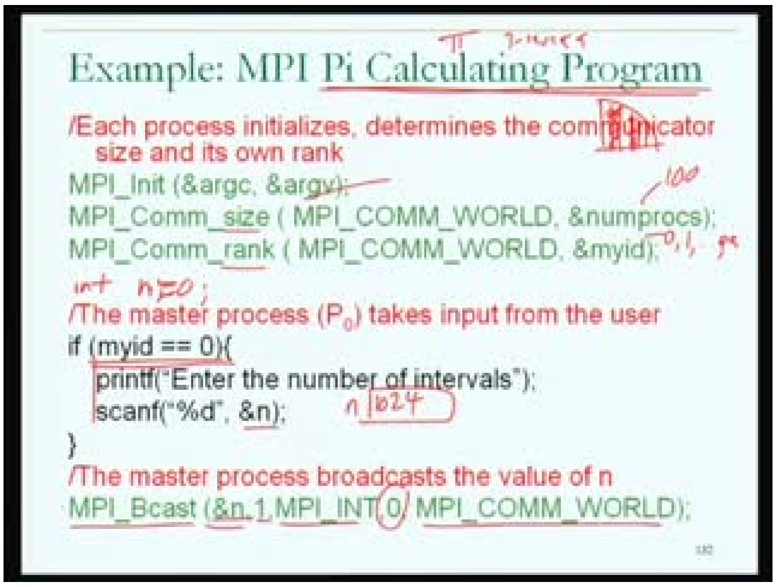
So, the in fact the size of this buffer in this particular example is going to be 100 multiplied by 10 multiplied by 4 bytes, so that is the size of the buffer.

You will know that each of the processes in the communication is going to send 10 integers corresponding to the ten elements of its variable data; data is declared in each of the processes as an array of size 10.

So, 10 multiplied by the 100, 10 integers multiplied by the hundred processes who are going to be involved in the gather, so once this data has been allocated the root process along with all the other processes, so note that what follows is going to be executed by all the processes in the parallel program to all of them execute MPI gather; in which they specify that the data in their local variable called data which includes 10 integers.

Let, supposed to be gathered into a buffer leading to and the size of the buffer is group size multiplied by 10 integers which is present in process with rank 0 of the MPI communication world, MPI communicator world.

(Refer Slide Time: 04:19)



Example: MPI Pi Calculating Program

```
/Each process initializes, determines the communicator
   size and its own rank
MPI_Init (&argc, &argv);
MPI_Comm_size ( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank ( MPI_COMM_WORLD, &myid);
int n=0;
/The master process (P0) takes input from the user
if (myid == 0){
   printf("Enter the number of intervals");
   scanf("%d", &n);          n 1024
}
/The master process broadcasts the value of n
MPI_Bcast (&n,1,MPI_INT,0, MPI_COMM_WORLD);
```

So, in this way it is possible for even more than one piece of data, more than 1 integer to be communicated using the same functions that we are talked about. Now let us proceed to look at a specific example of parallel computation, the specific example which I am going to look at is we are going to try to write or at least understand the structure of an MPI program which is going to do a calculation. And the calculation which I am going to refer to is actually to calculate the value of the constant pi what I mean by pi is pi which you know has a value of 3.141592565 whatever.

But, the purpose of this program is try to calculate the value of pi more accurately, is going to use some kind of numerical integration procedure. And we are not to concern about the actual procedure that the competition; that is going to do or we are more concerned about the fact that, it is going to be executed as a parallel program.

So, the work of computing pi is going to be divided among larger number of processes. Each of which will do some of the work towards computing pi, then they will all the data from, all them will be accumulated to actually get the final value of pi. And you want to understand more about how this could be done with a single MPI program.

So, we are going to set this out by writing the MPI code, I am going to show you the MPI code; we are now going to reason about how the calculation is done, just I want to the just bear in mind that we are going to be using some kind of a numerical integration technique.

So, we are going to be computing the area under one quadrant of a circle and that quadrant of the circle is actually going to be divided into multiple slices. Each of the processes will compute one of the slices of area possibly if the number of processes is low, each of the processes might compute more than one slice accumulate that area and send it back to the root process which will do the accumulation and therefore the computation of pi.

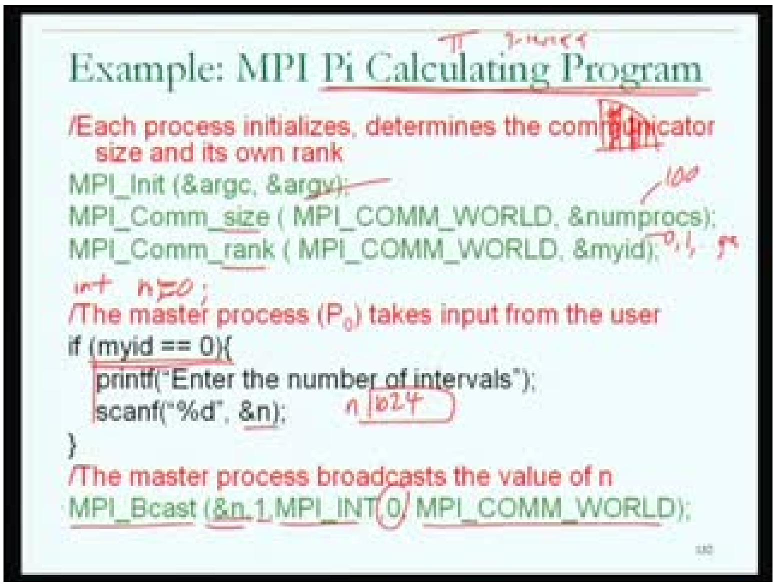
So, that is the nature of the computation and the actual statements in C which do that integration are not too important to us there included in some form, but please do not pay too much attention to that. Now, as in any MPI program of the kind that we are talking about where we are assuming that its written in an SP, MD mode, the same executable is going to be run on each of the processors, as each has a separate process. Therefore it will be the case that each process will have to determine the size of the communicator as well as its own rank and we know how to do that now.

So, the program starts with MPI in it and each process subsequently where find out, how many processes this program is running as and what its own identities; to the one process will receive a return value of if the program is running as hundred processes each of them will end up with a knowledge that there are hundred processes.

And of the hundred processes one of them will receive a value of MID equal to 0 and other will receive a value of MID equal to 1 and so on, one of them will receive a value of myid equal to 91.

So, each of them will have a clear understanding of its distinct rank, subsequently one of the processes which I am designating as a master processes we will take some input from the user. Now you should bear in mind that parallel programs like the ordinary programs that we write on to run on single computers; may have to interact with a user and therefore they may be some calls to get for example in additional information from the user, in this example we are going to get feedback from the user about how many slices we have to break do the integration over.

(Refer Slide Time: 04:19)



Example: MPI Pi Calculating Program

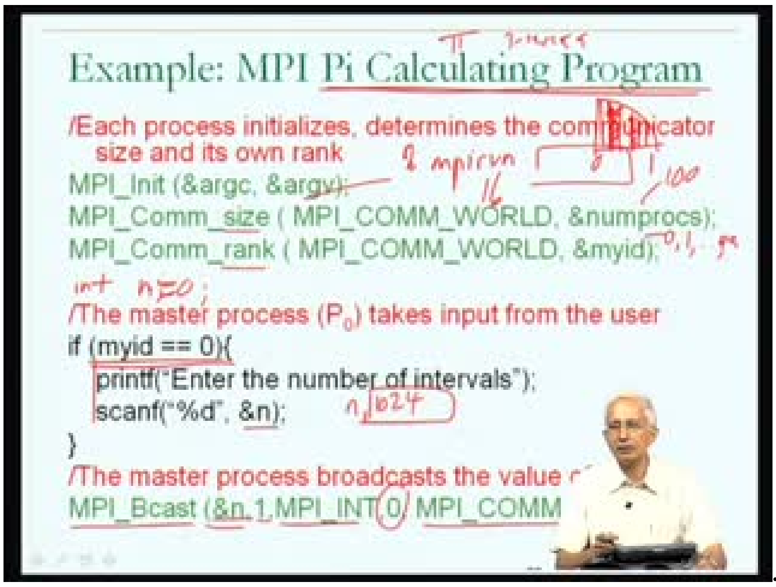
```
/Each process initializes, determines the communicator
   size and its own rank
MPI_Init (&argc, &argv);
MPI_Comm_size ( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank ( MPI_COMM_WORLD, &myid);
int  n=0;
/The master process (P0) takes input from the user
if (myid == 0){
   printf("Enter the number of intervals");
   scanf("%d", &n);
}
/The master process broadcasts the value of n
MPI_Bcast (&n,1,MPI_INT,0, MPI_COMM_WORLD);
```

So basically what happens is, the process known as P 0 is the only one which will find myid equal to 0, and it will print a message on the console of the parallel computer asking the user to type the number of intervals; that you wants to be, he or she wants to be use in the computation, this will then be read into a variable called n.

Now, the variable called n is a local variable in some sense of process P 0 remember that the processor the other processes will not know the value of n; therefore if process P 0 wants the value of n to be known across all the computer, all the processes in the computer in the parallel program then it would have to broadcast the value of n to all the processes in comprising the parallel program which is what it does next, it will broadcast the value of n to all the processes which constitute this parallel program. And if can do this using the MPI broadcast group function, so it mentions that again somewhere in here they should have been a declaration of the variable called n.

So, each of the processes has a variable called n, it just happens that the copy of n in the address space of process P 0 will have the value that is meant to be broadcast; for example, if the user had typed in that he wants 1024 intervals to be used, then the value of n in the address space of process P 0 will be 1024. Of course the value of n in the address space of other processes will have its, whatever it is a default value is possibly 0, but after the broadcast which is done by process P 0 as root using the value in its variable n which is 1 integer value to all the processes in communication world.

(Refer Slide Time: 09:21)

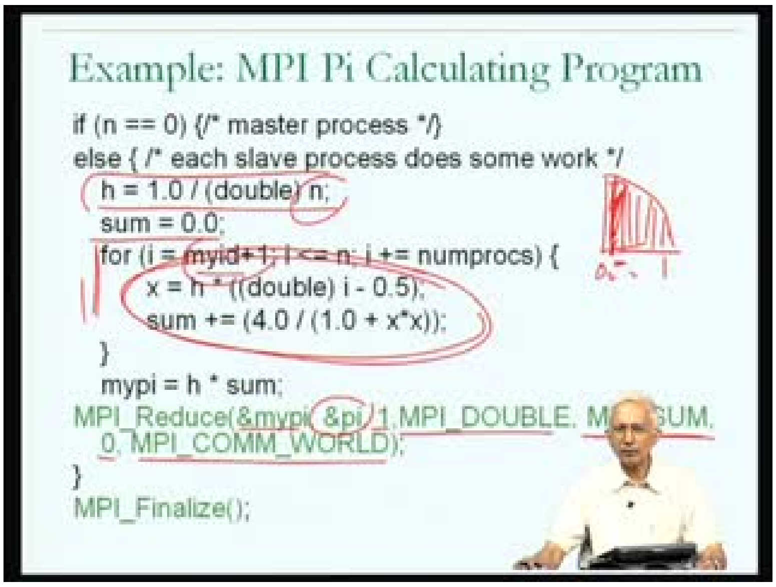


Example: MPI Pi Calculating Program

```
/Each process initializes, determines the communicator
   size and its own rank
MPI_Init (&argc, &argv);
MPI_Comm_size ( MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank ( MPI_COMM_WORLD, &myid);
int n=0;
/The master process (P0) takes input from the user
if (myid == 0){
   printf("Enter the number of intervals");
   scanf("%d", &n);        n 1024
}
/The master process broadcasts the value
MPI_Bcast (&n,1,MPI_INT,0, MPI_COMM
```

So, after this statement, after this function has been executed by all the processes which comprise my parallel program, the value of n on all the processes will be equal to 1024; that is what the broadcast function achieves. Subsequently we can go into the region of the program where each of the processes does its step of integration or its steps of integration. And the number of steps that each process will have to do, will depend on how many intervals the user specified as well as the number of process that this program is running else.

(Refer Slide Time: 10:27)



Example: MPI Pi Calculating Program

```
if (n == 0) {/* master process */}
else { /* each slave process does some work */
  h = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid+1; i <= n; i += numprocs) {
      x = h * ((double) i - 0.5);
      sum += (4.0 / (1.0 + x*x));
  }
  mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, M[illegible]SUM,
  0, MPI_COMM_WORLD);
}
MPI_Finalize();
```

Remember that this same parallel program could be run as 2 processes or as 4 processes or a 16 processes, the number of processes is not hardwired into the program; when I run this program using MPI run I have the option of specifying how many processes I wanted to run is and I could that time decide to run it this is in responds to the command prompt, I could decide to run it as 16 or 32 or 164 processes.

Now subsequently the user specifies that he wants the program to calculate the value of pi using a 1024 intervals; in other words dividing the area of the unit circle quadrant between 0 and 1 into 1024 intervals in doing the numerical integration, then clearly if I have 16 processes doing this work then each of them should do more than one of the intervals.

Therefore, within the region of the program that is going to do the work I am not including the master process, I am using all the other processes. So, that is why I will refer to the other processes of the slave processes, and each of them will do its work in the numerical integration. So, the way that this is being done is that it calculates the size of again remember while doing this calculation by quadrature the unit circle and calculating the area under this.

So, it finds out the slice size, in other words 1 divided by the value of n that is the actual width of this triangle, this region which is going to be, whose area is going to be calculated; it initiates sum which is the total area then it has computed in the slices that is deals with to 0. Subsequently it does some number of slices depending on what its id is, so if its id is 1 then it computes the second and some subsequent number of slices until it has computed slices reaching the end.

So, basically the different processes are picking consecutive slices the process number 1 will do as seen over here slice number 2, process number 3 will do slice number 4 and so on. And then after the last process has done its work, process number one will do the next slice and so on, that is what is being achieved by this integration.

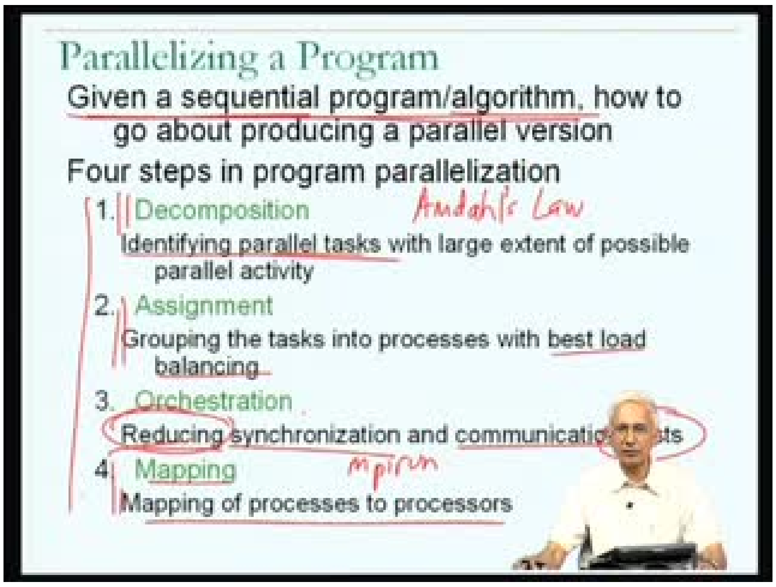
So, this type of integration I would not going to refer, but after we have reached this point, in the program each of the processes has done its calculation of its region of the this quadrant of the unit circle it can then calculate what its contribution to the value of pi is and now all the processes can communicate what they think, there contribution to the value of pi is back to the root process using MPI reduce.

In MPI reduce the nature of the computation is each of the processes is going to communicate its value of pi to the root process, and what is going to communicate is 1 doubles position value, because the computation that we talked about; it was clearly not computing an integer value. But a double position value the reduction is going to happen by adding all these values together and putting them into the variable called pi inside process P 0 the communication includes all the processes inside MPI-COMM-WORLD subsequently process P 0 can be written, so that it prints out of the value of pi.

In effect what we have achieved is and then the finalizing can be done what have we achieved, we have achieved a clear understanding of how we could write a parallel program which is actually computing the value of pi dividing the work of the computation across some arbitrary number of slave processes. And we have in some sense if we not understood the mechanics behind how the computation of pi is done, I will leave that is an exercise for you, but very clearly more complicated or even simpler parallel programs are well within the preview of what can be done using a small number of MPI communication primitives.

Now with this, understanding of MPI and we have demystified MPI, we understand that it is not too difficult to move from writing C programs to writing parallel programs which can be run using MPI parallel programs; that can run on large clusters of work stations or large parallel machines. There by reducing the amount of time that it would take to execute, let say a program that may have initially take of a huge amount of time on a sequential single computer.

(Refer Slide Time: 13:59)



Parallelizing a Program

Given a sequential program/algorithm, how to go about producing a parallel version

Four steps in program parallelization

1. Decomposition
   Identifying parallel tasks with large extent of possible parallel activity
2. Assignment
   Grouping the tasks into processes with best load balancing
3. Orchestration
   Reducing synchronization and communication costs
4. Mapping
   Mapping of processes to processors

Now with this we will move into a slightly more abstract mode, I want talk about MPI any more, but I will talk in general about the activity of parallelizing a program or writing a parallel application in terms of will run through a few examples of the kinds of decisions one may have to make along the way and so on.

Now the perspective that we are going to take on this is, that we have a sequential program or algorithm; and we are interested in understanding what has to be done to move from that sequential program or algorithm to a parallel version, of the algorithm a parallel program to achieve the same objective in a shorter amount of time, since we have a parallel computer.

Now, the general ideas if I by saying that we have a sequential algorithm, all that I mean is we have some understanding of how we would do or how we would solve the problem or achieve the objective.

If we had only one computer to achieve the objective on so, that is all that I mean by saying given a sequential program or a sequential algorithm, we are more concerned about given a specification of a problem. And we may have the specification of the problem by an understanding of how to solve it for a sequential computer, how do we go about producing a parallel program to do the, to achieve the same objective people will often talk about a four step procedure for writing a parallel program and they may talk about the four steps naming them as Decomposition, Assignment, Orchestration and Mapping.

And let me just give you a rough idea about these steps are just for completeness; now the objective of decomposition is to identify the parallel tasks which could be used as the parallel processes inside the parallel program, but essentially identifying the way in which we are going to break up the work in order to get the maximum possible parallel activity.

Now, just note that if in writing a parallel program its quite easy to say, let suppose I want to write a parallel program to compute the value of pi as we just did, its quite easy to say I will write a parallel program which runs just two processes. But as we saw in the previous example it is quite easy to write a parallel program which runs as 16 or 32 or 64 or some arbitrary number of processes to compute the value of pi.

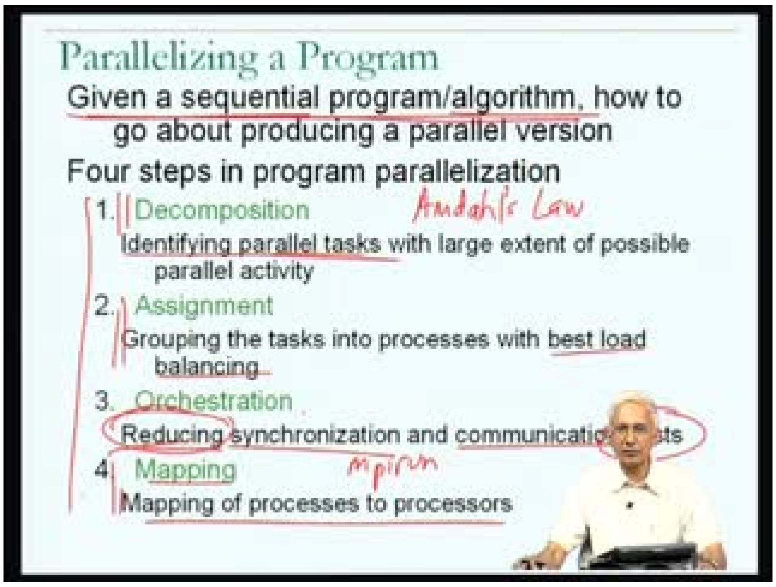
In general depending on what the application is they may be an appropriate number of parallel tasks that will be best for achieving the parallel objective and that identifying what those parallel tasks might be in the step called decomposition. May be the first activity to take place, rather than trying to think right from the beginning about the variables or the nature of the reduces and scatters etcetera.

At the first I am trying to figure out what is the nature of the parallelism and what might be the appropriate a parallel level of parallel the task activity, this is also important for us from the perspective of what we are seen in a previous lecture about Amdahl’s law which told us that essentially the speed up or the extent to which parallelizing a program might cause it to be speed it up. Is going to depend on the fraction of the sequential version of the program which could not be parallelized at all and this initial step will give us some idea about what that sequential fraction might be. Now, after we have got some perspective on how we would like to decompose the problem.

The next step is to actually do assignment which is grouping the tasks into processes, now in the first step we had explicitly not talked about processes, but rather about tasks. And once again the right way to think about this is, once again to think about our pi calculating program if you look back or think about the pi calculating program each of the processes was not computing 1 slice of the numerical integration task.

But rather many slices of the numerical integration task in thinking about the decomposition of the problem, I might in fact not of thought of over the processes at all, but I may have thought about the individual tasks that may have thought we can actually have 1000 task, 1024 tasks which could be done.

(Refer Slide Time: 13:59)



Parallelizing a Program

Given a sequential program/algorithm, how to go about producing a parallel version

Four steps in program parallelization

1. Decomposition
   Identifying parallel tasks with large extent of possible parallel activity
2. Assignment
   Grouping the tasks into processes with best load balancing
3. Orchestration
   Reducing synchronization and communication costs
4. Mapping
   Mapping of processes to processors

Now, subsequently when I think about the number of processes that might make sense for the application I might realize that having one thousand and twenty four processes may increase the over heads of running, this as a parallel program. Because, the size of the gathers, the size of the broadcasts will become a little bit more, and then we know that underlying anyone of those group communication primitives there are going to a large number of sends and receives.

And therefore, fewer the tasks the more efficiently those are going to take place, so taking this into account we may actually come up with a situation where we started off with the 1024 tasks as the decomposition of the pi calculating program.

But we broke it down into a smaller number of processes, because we have realize that this may actually lead us with a better balancing of load across a number of processes in terms of the distribution of communication work among the processors of the parallel computer system.

Now, once we have this picture of the parallel program in terms of a collection of processes is, we can then do what is called orchestration trying to reason about at what points in time, at what points in their calculation; will the processes have to be synchronized or what points in the computation will the processes have to communication with each other. And from this we can estimate what there overheads of the costs associated with the synchronization or the communication among the different parallel processes is going to be.

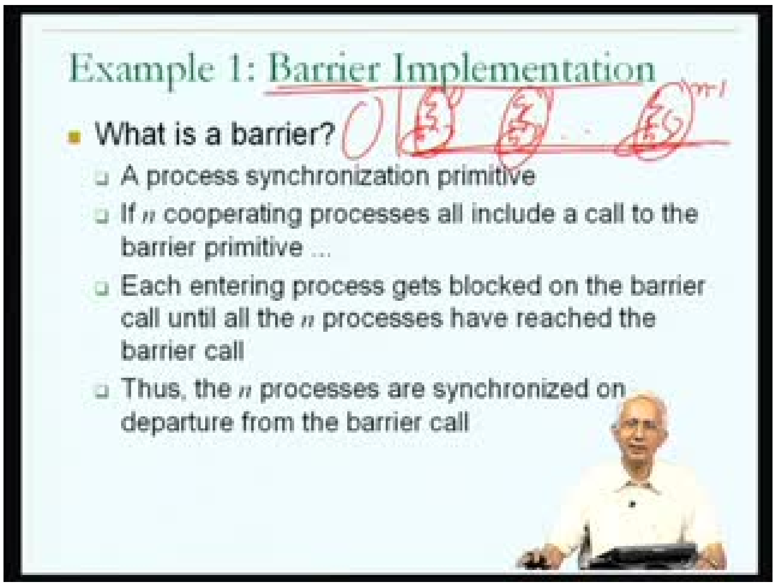
And in coming up with this idea what the overheads are going to be, we can make the effort of trying to reduce the overheads of those communications. For example, if at one some point we had been thinking about using point to point communication for a particular variable. And we realize that the communication is happening across all the processes, we may see that we could reduce the cost by using group communication but they may be other ideas that could come to mind in the step of orchestration.

So in orchestration, we move from a picture of what the processes are to how best to communicate, to set up the communication between the processes; in the last step of parallelizing a program people sometimes talk about mapping, and this is the step of the parallel programming where we actually relate the processes to the individual processors of the computer system.

So, mapping refers to the mapping of processes two processors, and this is not something that we talked about having explicit control in terms of our MPI program. You will recall that, when we talk about the MPI program the base assumption was the same program is run on all the processes. And we do not know, how many processes the program is going to run on, but in talking about initially causing the program to execute, we do have the capabilities of specifying exactly which and how many processors are going to be involved in the execution of the program.

And they may be other situations for other kinds of parallel programming scenarios where we actually have control over which processes are going to run on which processors; this is not immediately available to us in MPI in the form that is referred to over here and might not therefore be a typical step in writing in MPI program. Now, with this very general introduction, these are terms that you will see if you read books are tutorials on parallelizing a program I wanted to look at one or two specific examples in which I specify a task in activity, and then we will think about various possibilities about how it could be parallelized.

(Refer Slide Time: 21:01)



Example 1: Barrier Implementation

- What is a barrier?
  - A process synchronization primitive
  - If *n* cooperating processes all include a call to the barrier primitive ...
  - Each entering process gets blocked on the barrier call until all the *n* processes have reached the barrier call
  - Thus, the *n* processes are synchronized on departure from the barrier call

Now, the first activity I am going to talk about, is what is known as implementing a barrier; we have come across the term barrier when we talked about concurrent programming, I introduce the barrier as being a kind of synchronization primitive. We are also come across the barrier when we looked at the list of MPI group communication primitives. Because, the barrier is one of the MPI group communication primitives which we did not look at, I talked only about, I believe gather, scatter, reduce and broadcast.

But, one of the group communication primitive’s functions in the list that I put before you was in fact the barrier. So, ultimately the barrier is the name of a well-defined synchronization primitive.

And let me just tell you, what a barrier is? It is process synchronization primitive, it could be used in concurrent programming, and it could be used in in parallel programming. And the general idea is, if I have a situation where there are n cooperating processes and each of them includes a call to the barrier primitive.

In other words there are n processes process P 0, process P 1, through process P n minus 1, and each of them calls the barrier function or the barrier primitive, then each process entering the barrier. In other words each process which reaches call to barrier gets blocked on the barrier call until all the n processes have reached the barrier call.

In other words it is possible that process P 0 reaches the barrier function call first, but it will not proceed beyond the barrier function until all the n processes have reach the barrier function in their parallel programs. Which means that the barrier function call inside the parallel program constitute some kind of a line beyond which no process can proceed until all the processes have reached the barrier.

And that is why the name barrier seems to be a fairly good name for this function; it can be used to synchronize all the processes which are participating in the barrier. You should know that, I could a write a parallel program which has one additional process which is not participating in the barrier right. So, one could think about using a barrier to synchronize some number or some subset of the processes in the subset could include all the processes or could be less than that, depending on the needs of the parallel program.

So, it seems likely that the barrier synchronization idea is going to be quite useful in writing parallel programs, which is why it is included as a basic group communication primitive in MPI.

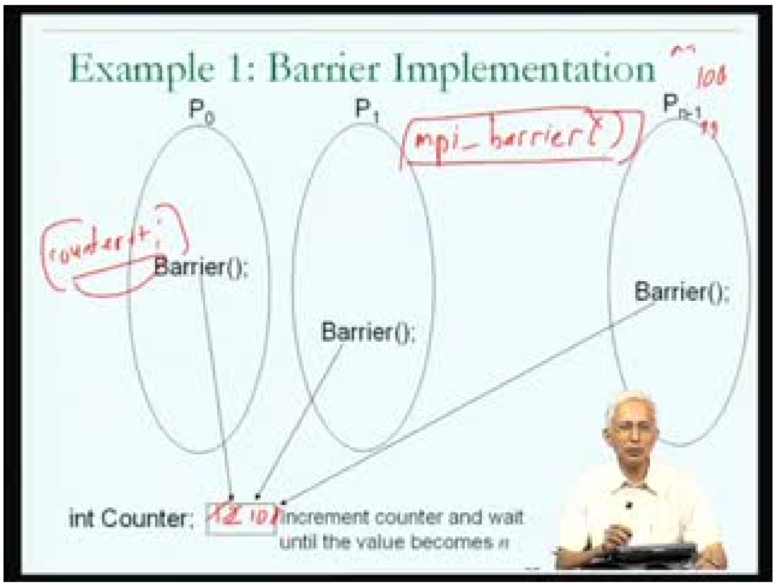
Now, why do I talk about implementing a barrier as an appropriate example through which to understand something about parallelizing applications, if we think about the idea of implementing a barrier, we could come across various base in which the barrier function. We are not talking about how you would implement the barrier function, we could think of various ways in the barrier function could be implemented. This is along the lines of how we talked about the lock primitive, we looked at a few possible ways to implemented lock, acquire lock and realize lock.

Some of our ideas were wrong, but we ultimately came up with ideas of how the lock, acquire lock and release lock functions could be implemented; you want to go through the same exercise for the barrier function.

So, the net effect of using a barrier is that the n processes which are synchronizing on a barrier will all depart from the barrier function call synchronized. None of them would have done anything other than having just come through the barrier, so each of them would have all of their work up to the point of the barrier and no more at the point that they proceed beyond the barrier.

Now, one simple idea for how we could implement a barrier remembers the barrier involves n processes; each of the n processes is ultimately going to call the barrier function. And we know that each of them will proceed beyond the barrier, only after all of them have reached the barrier.

(Refer Slide Time: 24:28)


Example 1: Barrier Implementation
100
P0
P1
Pn-1
99
mpi_barrier()
counter++;
Barrier();
Barrier();
Barrier();
int Counter:
1 2 10
Increment counter and wait until the value becomes n

Now, one simple idea of how we could implement the barrier could be that, we could use a shared variable to implement the barrier, and if we if the system that you are working on has shared variables, and very clearly a shared variable could be used to implement the barrier. Remember that, we used a shared variable to implement a lock, and a lock is a basic synchronization, mutual exclusion primitive there was nothing wrong in assuming that you have shared variables in talking about an implementation.

So, how could I use a shared variable to implement a barrier, the simple idea is something like this; I will have a shared variable which is I am going to be used as a counter. I will initialize the shared variable to 0, substantially when any one of the processes reaches the barrier function, what will happen inside the barrier function, in other words the body of the barrier function will be to increment the counter variable and then to wait until the value of the counter variable becomes equal to n.

So, within the barrier function I am going to find something like counter plus plus; and busy waiting on until counter becomes equal to n, so while counter not equal to n busy wait. So that is what I expect could be present inside the barrier function that is one possible implementation of the barrier function.

And you could know that it will work because, after if initially that value of counter is 0 after process P 0 reaches the barrier it will make barrier equal to 1, but it will continue to busy wait on the loop, because barrier has not yet become equal to n we can assume that n is equal to 100 so n minus 1 is equal to 99.

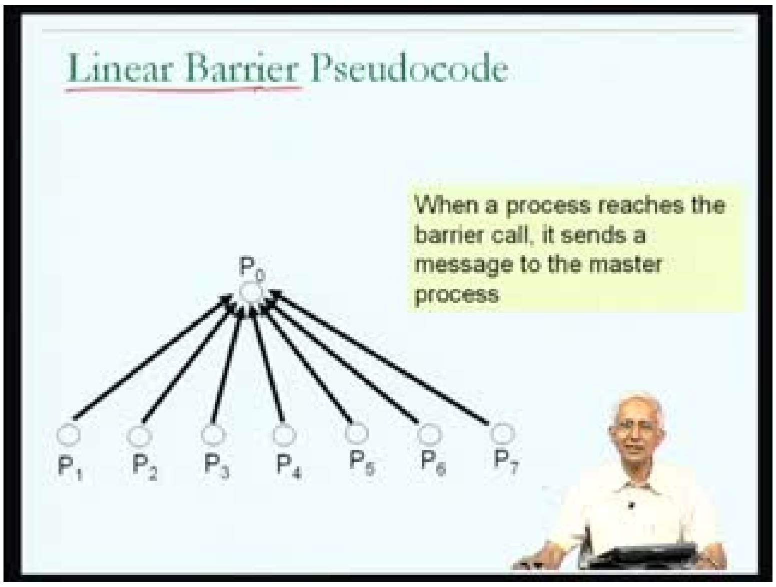
Subsequently one of the other processes reaches the barrier increments n to 2, it also finds that n is not, the counter is not equal to n or 100 and therefore it is also busy wait on its loops within the barrier function. And this keeps on going until the last of the n processes increments, the barrier to 100 at which the point in time all of the processes will find out that n has become equal to 100, and they will exit from the busy wait loop. And therefore, this is the perfectly legitimate way to implement a barrier if you have a shared variable system.

Now, the question that arises now is what if I have system in which there are no shared variables and I have to implement the barrier using message passing now that is the situation that we want to think about how would you implement a shared variable using message passing?

And you will now realize why this is an interesting and an important example. Because, but we are going to talk about now, is the various ways that MPI barrier function could conceivably have been implemented.

You will note that the MPI barrier function is available an MPI, but is ultimately written as a function within the MPI library, and is possibly written using MPI send and receive function calls. The more primitive kind of message communication primitives provided and supported by MPI.

(Refer Slide Time: 28:03)


Linear Barrier Pseudocode

When a process reaches the barrier call, it sends a message to the master process

And if there are many different ways of doing that then very clearly we would want to make show that MPI barrier uses the best possible way to implement the barrier, which is why looking at a few alternatives in terms of decomposition orchestration etcetera, might be useful for us to understand more about parallel programming in general.

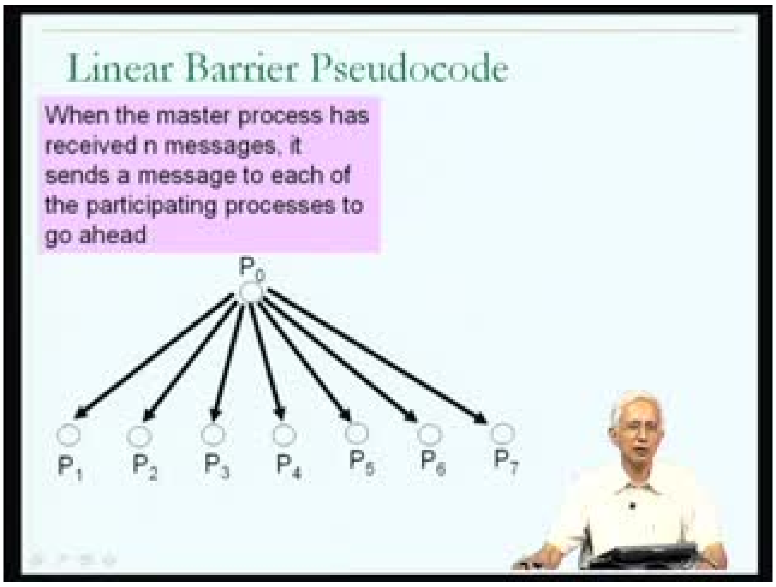
So, it is a simple example, but are fruit full one because, we will be able to understand the little bit more about how important it is for the people who write the functions like MPI barrier to think through the problem a little bit. Now, I am going to start by giving starting with the very simple idea of what is typically known as a linear barrier, we are going to write Pseudocode for it, now going to write the actual C functions or MPI functions we are going to use a pseudo notation to save some time.

Now, so the situation is that I am going to create the barrier between processes P 1 through P 7 using process P 0 as some kind of a master process, so process P 0 is not one of the processes participating in the barrier, but is used as a master process to create a barrier they can be a cross process P 1 through P 7.

So, the way that the, will start up by looking at the functioning of the barrier implementation pictorially. The way that the linear barrier is going to work is that, when each of the processes reaches its barrier the call to the barrier function within its program. It sends a message to the master process.

When a process reaches the barrier call, within its own body of within its own program that, what is happening inside the call to barrier is first the process sends a message to the master process. So the first process that reaches the barrier sends a message to the master process then the second, then the third etcetera.

(Refer Slide Time: 29:30)



Linear Barrier Pseudocode

When the master process has received n messages, it sends a message to each of the participating processes to go ahead

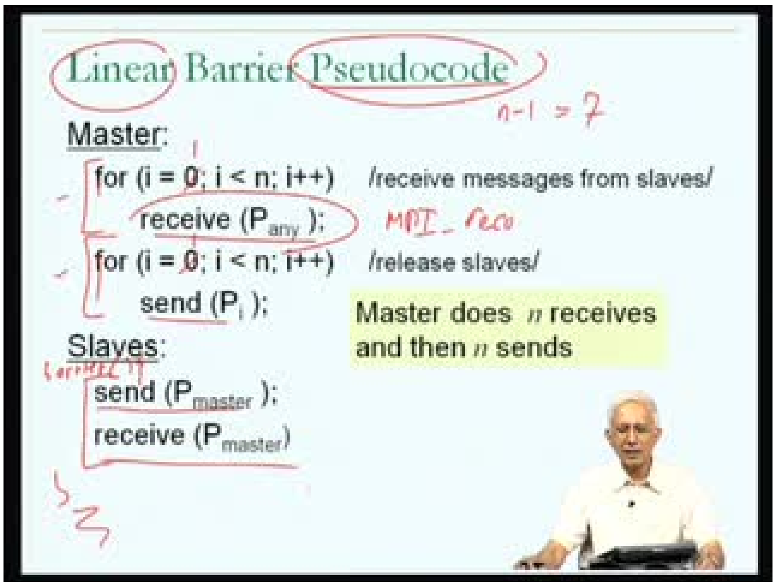
So, some point in time all the seven processes would have receive, would have reach the barrier and send a message to the master process. What should be happening within the master process? Within the master process when it receives a message from any one of the n processes in this case n is equal to 7, seven processes participating in the barrier, when it has received a message from each of the n processes participating in the barrier, it proceeds to send a message to each of those processes in return.

So what is happening inside the master is it is keeping track of the messages that it has received and as soon as that has received a message from each of the n processes it sends a message in return to each of the n processes this case 7.

So, it sense in order to process P 1 through P 7 and as soon as the process P 1 through P 7, each of them receives a message from the master it knows that it has finish with the barrier and can proceed to out of the barrier.

Now, this is how the linear barrier works, you can understand might it is called linear barrier; it is called a linear barrier, because in each of the steps that we talked about each of the processes, in the first step each of the processes sends a message to the master and in the second step the masters sends a message to each of the seven processes. And if this is happening in some sense in a sequential order as we were seen, and we will look at the Pseudocode.

(Refer Slide Time: 30:42)



Linear Barrier Pseudocode

$n-1 = 7$

```
Master:
    for (i = 0; i < n; i++)        /receive messages from slaves/
        receive (P_any);           MPI_recv
    for (i = 0; i < n; i++)        /release slaves/
        send (P_i);
Slaves:
    send (P_master);
    receive (P_master)
```

Master does n receives and then n sends

Now, what we mean by Pseudocode is, we are actually going to look at the statements the C like or MPI like statements are going to be executed by the master and by the slave. To the sequence of events is we said as soon as each of the slaves reaches the barrier it will do a send to the master.

So, if I looked into the barrier function this is what I will see in the slave, but it enters the barrier it does a send to the master. What does the master do? The master is waiting to receive a message from each of this slave. Therefore, within the master we expect that as part of the implementation of the barrier there is going to be a four loop which is going to be executed in our example seven times.

And each time through the four loop it is going to receive from any of the slaves, it does not matter which of the slaves it receive some; important thing is that it executes receive seven times and that each time through the loop is going to receive from one of the slaves.

So, could be the process P 3 finishes reaches a barrier first in which case it sends the message to the master, and the master receives the message right away. And we know that this is possible using the MPI primitives, primitive send and receives by having a receive from any source kind of a call.

Now, after the master exits from this loop we know that it has received a message from each of the participating processes in the barrier; it can then go ahead to a second loop, where it sends a message to each of the processes participating in the barrier. So, it sense first to process P 0 then to P 1 then to P 2 up to first the process P 1, so this would be corrected to 1 it does not send to in our example the processes not participating in the barrier itself, so this should be connecting also corrected to 1.

So, each of these loops is excluding n minus 1 times, where n minus 1 is what is equal to 7. So then the process, the master process sends a process a message to each of the slave processes; what happens in the slave, the slave just receives the message receive from the master.

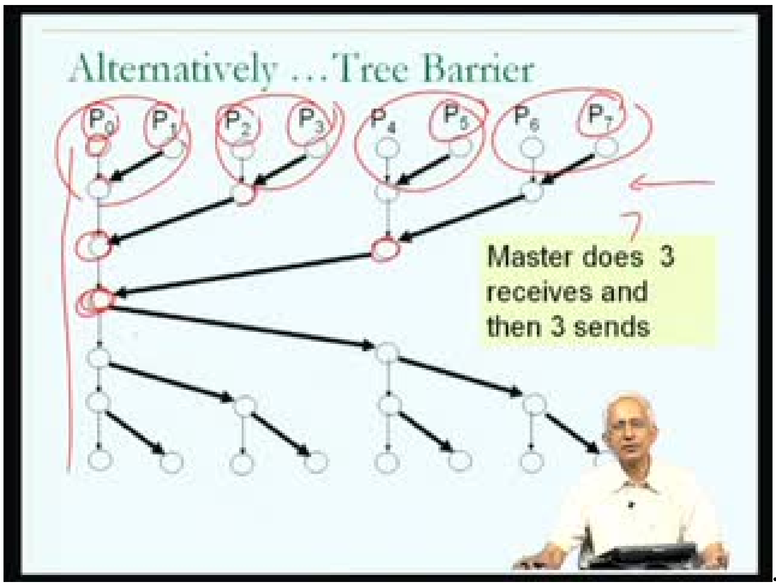
Therefore, what we are looking at down here is the implementation of the barrier function as far as each of the slaves is concerned; this is what the slave is going to execute, the slave executes when it reaches the barrier it sends to the master, and then it receives from the master.

And soon as both of these calls have finished, it is done with the barrier and proceeds to the next part of its program and it is synchronized with the other processes synchronizing on the barrier. Why is it called a linear barrier? You will notice that the amount of time that it takes for this barrier to work is going to be determined by looking at the master, the master first has to go through the upper loop seven times or n minus one times in general.

Then it has to go through the lower loop again seven times there n minus one times, in general and therefore this is a procedure in which it is linearly or sequentially going through each of the child processes receiving from one of them, each time through the first loop and sending to one of them each time through second loop, which is why it is called a linear barrier.

And you should notice that we could quite easily encode this in MPI by just replacing receive by the appropriate MPI receive etcetera; quite easy to map this pseudocode I called it pseudocode, because I have not actually used MPI sends and receives here, but I am using a more general version just like pseudocode in c. So, we understand how this linear implementation of the barrier could work and further we understand that it should be easy to do better than this.

(Refer Slide Time: 34:07)


Alternatively …Tree Barrier

Master does 3 receives and then 3 sends

Let me give you simple idea, for suppose that in in this version of the barrier I am trying to synchronize all the eight processes from P 0 through P 7 with each other. In that you which I am now going to do are, I will avoid making the master explicitly, I will avoid having the process P 0 as the explicit master as far as all the processes are concerned by distributing the communication that each process will do when it reaches the barrier.

So, what we are going to see from this point on is the kind of messages that each of the processes will send when it reaches the barrier. So, you will notice the what happens, when process P 1 reaches the barrier is that process P 1 send a message to process P 0, when process P 3 which is the barrier, it sends a message to process P 2 and so on.

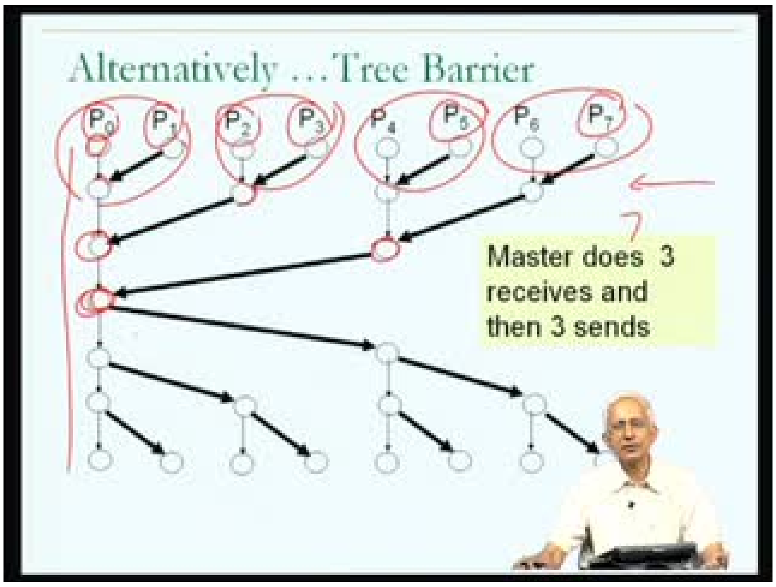
So, we are talking about four messages that are send, when process P 0, P 3, P 5 and P 7 reach the barrier. What is process P 0 do when it reaches the barrier, it does a receive for a message from process P 1, what is process P 2 do when it reaches the barrier, it does a receive of a message from process P 3.

Therefore, what we are seeing in this line of four message sends and four message receives is that process P 0 and P 1 have both reach the barrier by the time the send and the receive have completed. Same is true as far as P 2, P 3, P 4, P 5, and P 6, P 7 are concerned.

Now, in the second we have not yet finish with our implementation of the barrier; but in the second step of implementing the barrier will try to synchronize or spread the information about which processes have reach the barrier by causing process P 2. Once it has come out of the receive of the message from P 3 to send a message to process P 0 what is this going to achieve? This is going to achieve the equivalent of process P 0 after having reach this point it then there is a receive a message from process P 2.

So, by the time process P 2 has finished with the send receive interaction with process P 2 it is known that P 0, P 1, P 2 and P 3 have all reach the barrier. And a similar situation would have been received as far as P 4 through P 7 is concerned. Now, with one more communication it is going to be possible for process P 0 to become aware of the fact that all the processes have reach the barrier; and that message will have to be sent from P 4 and received by P 0.

(Refer Slide Time: 34:07)



Alternatively ... Tree Barrier

Master does 3 receives and then 3 sends

So, with a total of 4 plus 2 plus 1 or seven message sends and seven message receives, we have been able to inform process P 0 that all the processes have reach the barrier, Subsequently all the processes have to in turn to be informed that this information.

Therefore, we will have, they will have to be another around of communication in which process P 0 first informs process P 4 and then process P 0 informs process P 2 and process P 4 informs process P 6 in effect the reverse what happen in the first round of communication. But, so on all the processes will have receive the information that we had been accumulated into process P 0 at this point in time.

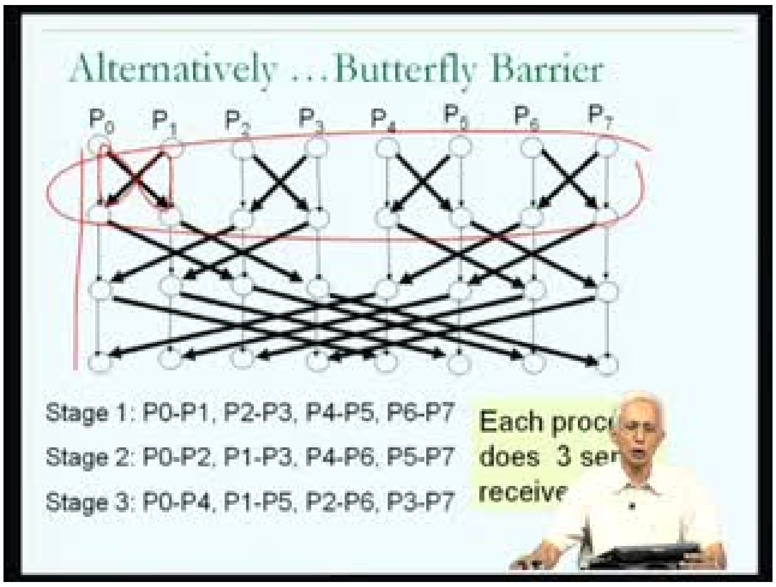
Now, if you look at this you may see that the number of messages that had to be sent has not really come down from what had to be done in the case of the linear barrier, but what has been achieved is that is no longer necessary for process P 0 to receive a message from each of the other processes in a linear fashion. And the initial communication between P 0 and P 1 could happen in parallel with the initial communication between P 2 and P 3 which in turn, could happen parallel between the other initial communications.

Therefore, allowing the parallel nature of the parallel machine to be exploited causing a reduction in the total amount of time that it takes to actually achieve the barrier synchronization.

So, we may not have reduce the number of messages that have to be communicated we still have seven messages to be communicated in the upper half and seven message to be communicated in the lower half. But the amount of time for the message communications to happen has been cut down, because of the parallelism that is exploited.

Now, when you look at this diagram or when you look at half of this diagram you could immediately come up with a name for this kind of an implementation of a barrier. Because, what we see of over here is something that looks very much like the tree that we talked about not too long ago, and this implementation of the barrier is often known as a tree barrier. This also leads us into some possible improvements of the idea of the barrier implementation beyond this.

(Refer Slide Time: 38:26)



Alternatively …Butterfly Barrier

Stage 1: P0-P1, P2-P3, P4-P5, P6-P7
Stage 2: P0-P2, P1-P3, P4-P6, P5-P7
Stage 3: P0-P4, P1-P5, P2-P6, P3-P7

Each proc
does 3 se
receive

For example, we could imagine that we could more aggressively overlap the way that the communication is happening conceivably within the first round rather than process P 0 and P 1, process P 1 sending a message to process P 0 and process P 0 just sending a message to process P 1 in the second half of the communication.

You will notice that process P 0 communicated within first step of the communication, and p 1 send a message to P 0 and P 0 received it. And then much later in the communication P 0 communicated to P 1 and P 1 received it.

So, what we are going to try now is, we are actually going to do the sends and receives between P 0 and P 1 overlapped in time. So, we are going to try to overlap the second half of our communication structure from the tree barrier with the first half of the communication structure.

So, we have communication a send from P 0 to P 1 and a send from P 1 to P 0 happening at the same time and there in fact happening at the same time as a sent from P 0, P 2, to P 3 and a sent from P 3 to P 2 and so on.

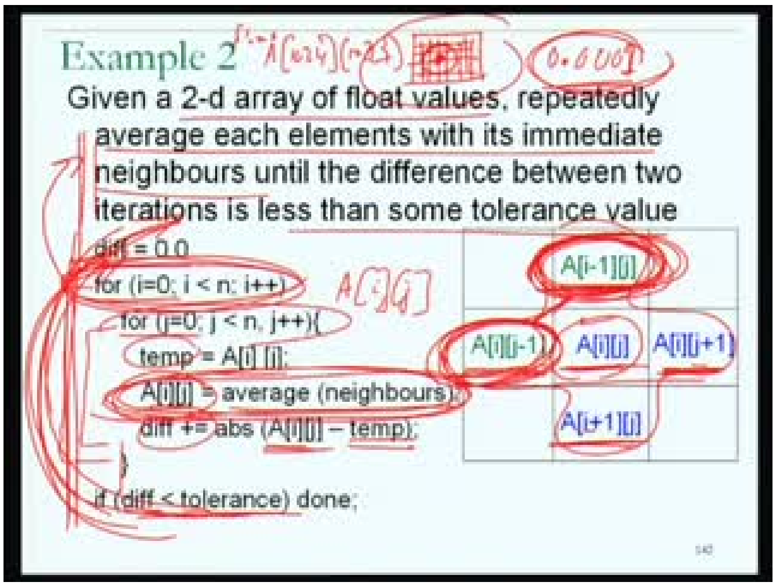
So, we are successfully overlapping the second half, the lower half of our tree implementation of the barrier; with the upper half of our tree implementation of the barrier. This will of course complicate the second step of the communication, you will remember that the case of the tree barrier in the second step of the communication, there was some communication between P 0 and P 2 that will now still have to happen, but we will talk about communication not only between P 0 and P 2, but also between P 2 and P 0 and at the same time between P 1 and P 3 and P 3 and P 1.

While the other half of the processes are also doing a similar communication and in the final step of our synchronization P 0 communicates with P 4 which is what was happening in the third step of the upper half of our tree barrier. And at this point in time when all the processes have successfully done the sends and receives the barrier implementation has been achieved. If you now count the number of messages, that had to be send and compare with the amount of time that is going to take you realize that this is going to be a substantial improvement over the tree barrier.

Now, what would you call this barrier; if you look at the diagram that has resulted at least at the first half of the diagram, you will understand by people tend to call this implementation of the barrier, a butterfly barrier these shapes look vaguely like butterflies.

So, in effect we would not be too surprised if we looked at the MPI barrier implementation, if we found something more complicated than the linear barrier that we first thought off. Because, subsequent thinking about the possible ways that the communications between different sets of processes could be overlapped, gave us inside into how we could come up with the barrier that could be substantially more efficient in terms of the amount of time that it takes for the communication to be achieved. So, in that sense this was useful exercise to better understand how the different steps of paralyzing a program may benefit from careful thought.

(Refer Slide Time: 41:22)



Example 2

Given a 2-d array of float values, repeatedly average each elements with its immediate neighbours until the difference between two iterations is less than some tolerance value

```
diff = 0.0
for (i=0; i < n; i++)
  for (j=0; j < n; j++){
    temp = A[i] [j];
    A[i][j] = average (neighbours);
    diff += abs (A[i][j] – temp);
  }
if (diff < tolerance) done;
```

Now, I want to do one more example of a parallel program, thinking about a parallel program; and this is slightly more realistic and that is going to try to do a numerical task of a particular kind. Now, the setting is that we have an application in which it is necessary to do some kind of an iterative procedure on a two-dimensional array of floating point values.

Remember that we talk about two-dimensional array as a data structure which contains many rows of data each of which contains many columns of data; it might be declared using for example, if it is a large 1024 by 1024 situation with 1024 rows and 1024 columns you can declare it and see using this kind of a declaration if the floating values and might declared a float a 1024 by 1024.

Now, what the application has to do is that repeatedly in other words it is going to have some kind of an iterative loop, each time through the loop it has to average, find the average for each of the elements inside the array computing the new element

For the new value, for that particular element of the array as the average of the value in that element, with the values in its neighboring elements and it will continue doing this iterative procedure until the difference between two iterations as far as the values in the entire array are concerned is not too much. And will quantify that saying that the difference between the value the two iterations is such that the total difference of the sum of the values of the total differences between the two iterations is less than some predefined tolerance value.

By what I mean by tolerance value is, some predefined value may be I will set the tolerance value to 0.0001 so that is a constant. So, this is procedure which is going to happen repeatedly until the values have been average in such a way, that they do not differ from one iteration to the next what differ by a negligibly small value.

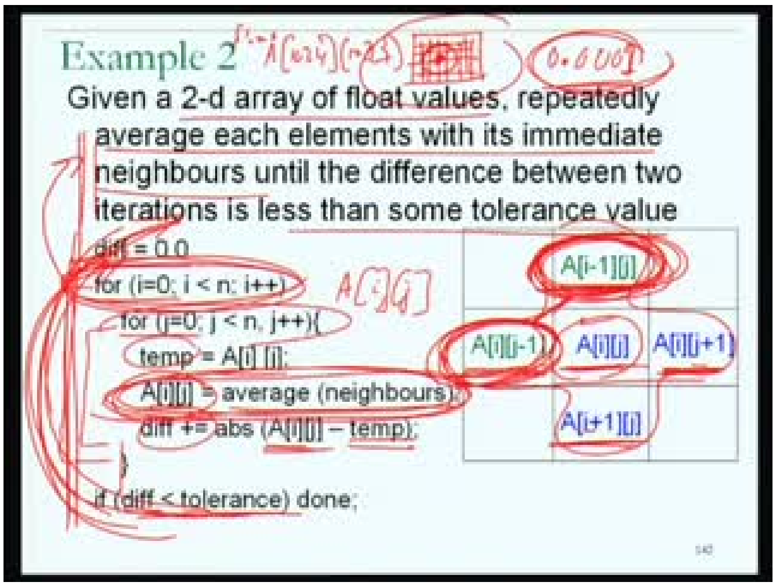
Let us just get a better field for what the averaging involved is; now if i was to try to write the C version sequential version of code to satisfy this requirement, I might do something like this.

So, what am I doing each time through this nested for loops I am computing the new value for one of the elements of the array I am computing the value for A of i comma j. Now, how is the new value of A of i of j to be computed? It is to be computed by averaging the old value with values of its immediate neighbors so I take the old value of A of i j and then, I compute the new value of A of i j by computing the average of the value of A of i j and its neighbors.

So, this is going to be the new value of A of i j and since I know that I have to repeat this procedure until the difference between two iterations has become less than some predefined tolerance value. I compute what is the difference between the old value of A of i j the, old value of A of i j which I had stored in the variable call temp and the new value of A of i j.

So, I compute what the difference is and I accumulated in a running sum and I had initialized the running sum to 0 and at the end of the iteration I check if the total difference between this iteration. And the previous iteration was less than my tolerance value possibly 0.0001 then I know that I am done otherwise I proceed to go through this iteration once again.

(Refer Slide Time: 41:22)



Example 2

Given a 2-d array of float values, repeatedly average each elements with its immediate neighbours until the difference between two iterations is less than some tolerance value

```
diff = 0.0
for (i=0; i < n; i++)
    for (j=0; j < n; j++){
        temp = A[i] [j];
        A[i][j] = average (neighbours);
        diff += abs (A[i][j] – temp);
    }
if (diff < tolerance) done;
```

A[i-1][j]
A[i][j-1]
A[i][j]
A[i][j+1]
A[i+1][j]
0.0001
A[i][j]

So, this is the iterative procedure involves repeating this w nested loops over and over again until this condition is the determination condition is satisfy.

Now, to better understand what is going to be involved in parallelizing this activity, we need to look more carefully at what the competition step is and this is the underlying competition step. So, let me draw a small picture which shows you, A i j as well as the immediate neighborhood of A of i j, so I am not going to show you the entire array, I am just showing you A of i j and its immediate neighborhood.

So, what is the immediate neighborhood of A of i j, the immediate neighborhood of A of i j will include the previous row, but not the entire previous row and the next row as well as the previous column and the next column.

Now the averaging which I am going to talk about is actually going to look at the neighborhood of A of i j which includes the element which is immediately above it in other words the row above it, but the same column as well as the element which is immediately below it in other words the same column, but the next row etcetera. So we are talking about in terms of the diagram the element which is to the north to the south to the east and to the west, of A of i j and the way that the averaging is going to be done is to add A of i j to A of i minus 1 j add A of i plus 1 j add A of i j plus 1 add A of i j minus 1.

And divide that sum by 5 and that is what would become the new value of A of i j repeatedly average each element with its immediate neighbors so I compute the sum of these 5 values divided by 5 and that is what I will store as a new value of A of i j.

Now you will notice that, I have put the values of the different elements which have being used in computing the average in two different colors; I have shown A of i minus 1 j in green, I have shown A of i j minus 1 also in green, but the other three I have shown in blue and you may wonder, why I have done that?

Now if you look at this sequential implementation of the application which satisfies our requirement, you notice that the innermost the inner loop is stepping I have this structure where I step through for A i equal to 0 then i step through i j equal to 0 that is how I came to i j.

But by the time I come to a of i j you will notice I would already have run through this procedure for A of i minus 1 j, as well as for A of i j plus 1 because A of i j plus 1 is on the same row is A of i j, but on a previous column which means it would have been the previous iteration of the for loop.

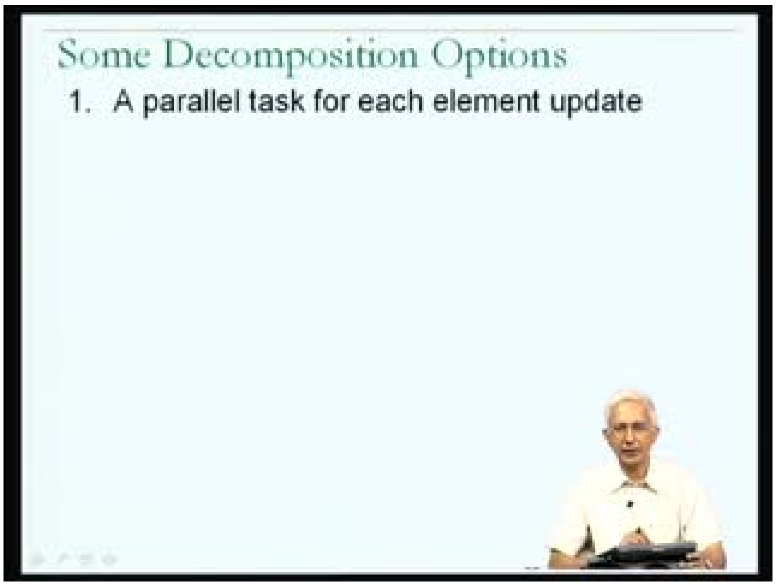
Similarly, A of i minus 1 j is on the previous row and would therefore have been it handled in the previous iteration of the i loop. Therefore, among the values which I am adding together to compute the average two are involving values which have been computed in this iteration of the outer loop not the i loop and the j loop, but the outer loop which is repeatedly being executed until the tolerance value is satisfied.

Whereas the other three are using the whole values of the array elements, you will notice that each time we iterate through i equal to 0 to n and j equal 0 through n and complete that, that is what I refer to as one iteration of the application of one iteration at the level of deciding when we have to stop doing this thing.

So two of the values that I am using in the competition or using the current iteration values of the array element I, i j, but the remaining three are using the values from the previously computed or the previous iteration. Hence, I show one the two elements which are using the current iteration values in green, and the three values which are from the previous iteration in blue.

And in talking about the parallel implementation, I will try to have the same interpretation of how to do the averaging and that is obviously going to cause some synchronization requirements to arise as for as the parallel application is concerned.

(Refer Slide Time: 48:50)

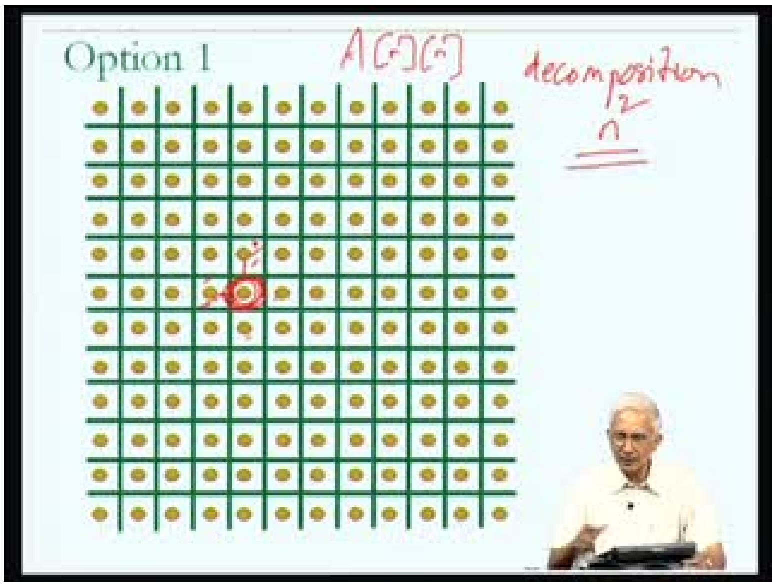


Some Decomposition Options

1. A parallel task for each element update

So, what we are going to look at its, look at a few decomposition options as far as how to decompose the work that has to be done across the various elements of the array; across tasks which could then be packaged into processes which could then be set of for communication etcetera.

(Refer Slide Time: 49:23)



Option 1

Now, one possible decomposition is that, I could have one parallel task for each array element; remember there are potently 1024 multiply by 1024 array elements. And pictorially what this would mean is here, I am showing you the entire array a the some number of rows, some number of columns, for the moment will not to concerned about the number of rows and columns; except that I will refer to the number of rows and the number of columns as n will assume that there n rows and n columns.

So, pictorially if I am going to have a separate task for doing the competition for each array element then there are going to be n squared tasks one task for each of the array elements and this is one way to think how I am decomposing. So pictorially, this is what the decomposition that i am thinking about, one task for each array element, now the first question that you will arise is, what kind of synchronization is going to be required? What is the maximum parallelism that is possible?

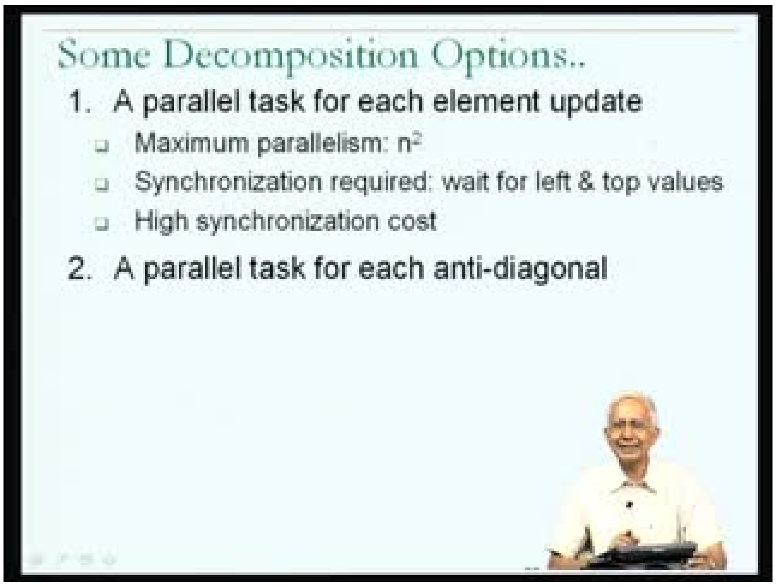
And you will immediately realize that I could package this as n multiplied by n processes and therefore the maximum parallelism possible. I could say, could be as much as n squared I could have n square processes which are at the same time progressing towards achieving the parallel task.

But, I do also have to worry about the kind of synchronization that will be required between the parallel processes and if I think about parallel process.

If I think about the competition which is happening for let say array element i j, I have realize that there is a need to synchronize with at least two of the other array elements, because it was important that in computing the value for A of i j, i used the newly computed value for the element to its north and the element to its west. Therefore, there was some synchronization that had to be done in order to achieve the same competition that we had in the case of the sequential program.

I am talking about the green values they have to be used here, not the blue values not the values from the current iteration, but the values from the previous, not the values from the previous iteration which is what I would use for the blue values. But the values from the current iteration therefore, there is a need for some synchronization, the competition for A of i j should not happen until the competition for the element to its north and to its west have completed and the synchronization must be included in the parallel program.

(Refer Slide Time: 51:21)



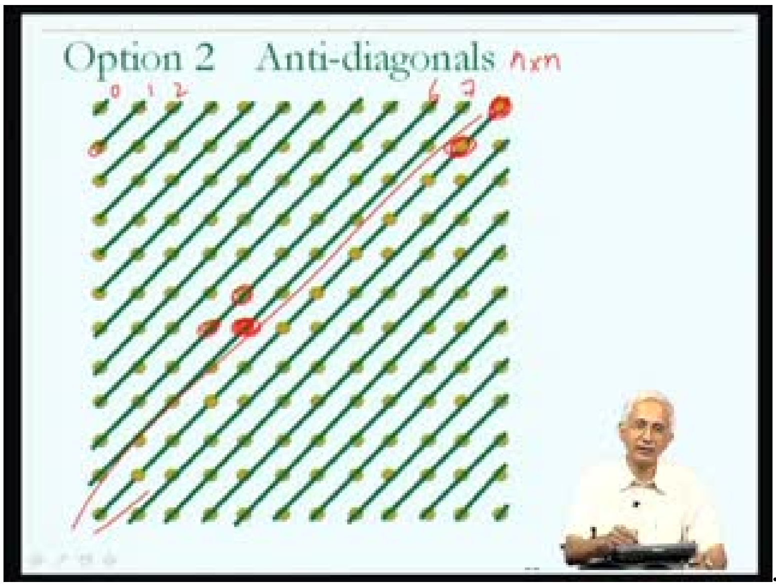
Some Decomposition Options..

1. A parallel task for each element update
   - Maximum parallelism: n^2
   - Synchronization required: wait for left & top values
   - High synchronization cost
2. A parallel task for each anti-diagonal

So, while the maximum parallelism might be n squared there is going to be the need for some synchronization if I use this decomposition option. So in analyzing this decomposition option quickly I might say that the maximum parallelism is very good, I could actually have an extremely parallel program.

But there is the need for a synchronization the competition for the process i j, is going to have to wait for the process to its left and top which I was calling west and north, little while back have completed. And therefore, that synchronization is going to have to happen and that is going to fair amount of synchronization, because that is approximately half of the it is going to have to synchronize with 2 of 5, 2 of the 4 other processes that it is relating to therefore, that amounts to fair amount of synchronization.

(Refer Slide Time: 52:11)


Option 2 Anti-diagonals

And I have to qualify this by saying that it is a high synchronization cost. Therefore, we do need to look for alternatives now one alternative that you could think of is to have what I will call a parallel task for each anti-diagonal; let me explain what I mean by an anti-diagonal. So, once again this is the n squared elements for which competition has to be done what I mean by in anti-diagonal is one of the lines parallel to the line that i show over here.

So, this is the big anti-diagonal and these are some other anti-diagonals, so the general idea that that I am working with now is that I will have one parallel task for each of these anti-diagonals. In other words I might have process P 0 computing for that anti-diagonal process P 1 computing for this anti-diagonal, what do I mean by that I mean that process P 1 will do the competition for both this array element, and for this array element process P 2 will do the competition for all these three array elements and so on.

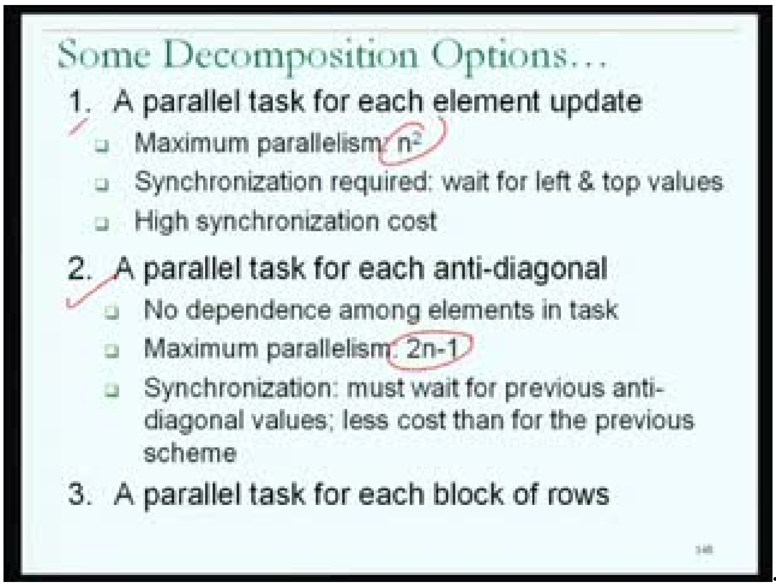
Now, what is the reason behind doing this, if you think about array element i j it is going to be computed by some particular process say, process seven. Now if I look at the diagram I realize that the computation for this array element had to synchronize with, computation for the array element to its north and the computation for the array element to its west.

And in terms of anti-diagonals both of those array elements were computed by one process and therefore in terms of the amount of synchronization that has to be done, if I decompose the task along these lines then it the amount of synchronization that will be required for array element i j is going to involve just waiting for the computation to complete for the anti-diagonal; which is before it which is much less than the amount of synchronization that had to be done in the case of the previous decomposition, what about the extent of parallelism?

In the previous decomposition I had potentially n square parallel tasks. In this situation I clearly have much less than n squared parallel task in fact, if the matrices were of size n by n then the number of anti-diagonals is in fact going to be 2 n minus 1 that you can confirm.

And further it could confirm that all the computations for any one of these anti-diagonals are actually independent of each other, in other words in computing this array element it does not have to worry about, this array element at all. Because, the computation of this array element, looks at its neighbors, which does not include this array element. And therefore, the computations within any one of these anti-diagonals are independent of each other, further the amount of parallelism is less than the case of the first decomposition.

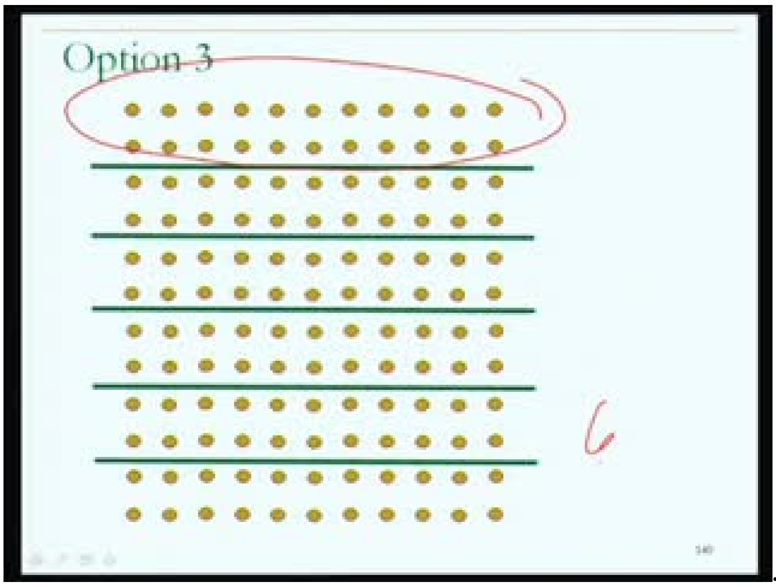
(Refer Slide Time: 54:23)



Some Decomposition Options…

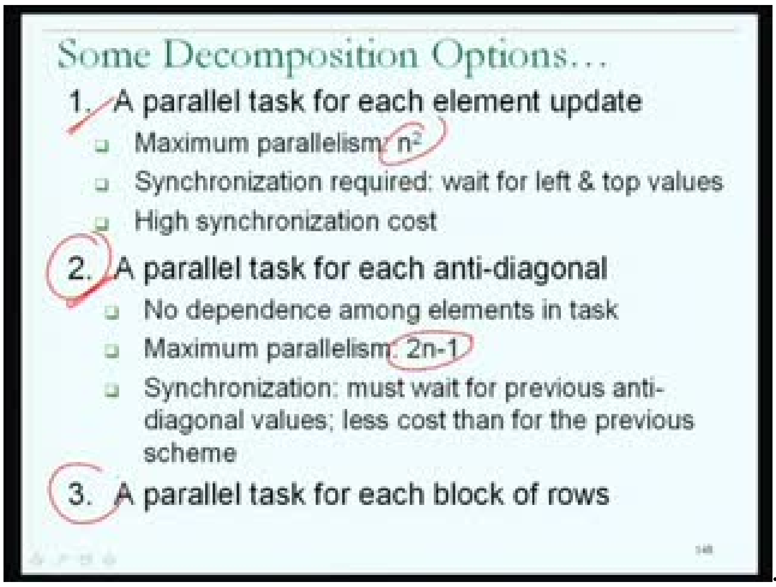
1. A parallel task for each element update
   - Maximum parallelism: n^2
   - Synchronization required: wait for left & top values
   - High synchronization cost
2. A parallel task for each anti-diagonal
   - No dependence among elements in task
   - Maximum parallelism: 2n-1
   - Synchronization: must wait for previous anti-diagonal values; less cost than for the previous scheme
3. A parallel task for each block of rows

(Refer Slide Time: 54:46)



But still reasonably significant the amount of synchronization is much less than in the previous option, and therefore this seems to be a superior decomposition to the first decomposition that we had talked about element by element. And one could proceed to look at the other decompositions for example you could think about a parallel task for each block of rows, what I mean by each block of rows this is a much simpler decomposition then the anti-diagonals I just look at the computation all of these array elements is happening on one process and so on.

(Refer Slide Time: 55:10)



Some Decomposition Options…

1. A parallel task for each element update
   - Maximum parallelism: n^2
   - Synchronization required: wait for left & top values
   - High synchronization cost
2. A parallel task for each anti-diagonal
   - No dependence among elements in task
   - Maximum parallelism: 2n-1
   - Synchronization: must wait for previous anti-diagonal values; less cost than for the previous scheme
3. A parallel task for each block of rows

So, I have 1, 2, 3, 4, 5, 6 parallel tasks and you notice that the amount of maximum parallelism possible is much less than 2 n minus 1 are obviously that n squared and possible the amount of synchronization has to be evaluated. So, the bottom line is that in given a parallel application and moving towards, given a sequential problem a sequential version of how to solve a problem in moving towards the parallel version of a solution to that problem, but has to preference look at various options and analyze them and figure out which of them makes the most sense before actually think about writing the parallel program.

And with this some briefs some what high level example, I hope that you have some feel should be fairly clear that the second option is superior to the first and some analysis would have to be done to place the third option; into the perspective of how good it is relating to the other two options.

So, with this I would like to wrap up our discursion of parallel programming by pointing out that the activity of writing a parallel program will require a good understanding of the underlying problem. And of possible a sequential implementation of the underlying problem, but should take into account the various aspects of parallelism in terms of the overheads that might be induced by certain decision such as decomposition, before proceeding to an encoding or writing the parallel program possibly in MPI.

With this we have finished our discussion of parallel programming, and in fact we have finished our discussion of all the topics in the syllabus of the course I hope it was of use to you and that change the way that you write your programs make the more efficient, thank you for participating in the course.