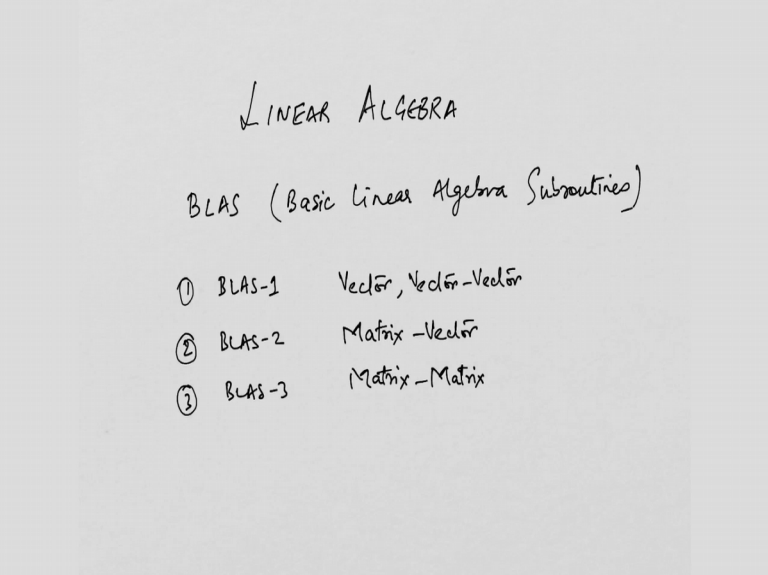
Introduction to Parallel Programming in OpenMP

Dr. Yogish Sabharwal
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture - 07
Basic Linear Algebra operations in OpenMP

(Refer Slide Time: 00:27)



Today, we will have a look at some linear algebra routines and how do you do various kinds of linear algebra using openmp; how do you paralyze these codes. So, as you know these codes like solving a system of linear equations, matrix operations these occur in a lot of scientific computations. We will focus on some simple operations in this course and then we will see how we can paralyze these operations using openmp.

So, let us start with something very simple at the most basic level. There are three different categories of linear algebra operations. So, there is actually a library called BLAS; basically in algebra subroutines and this is heavily used in a lot of scientific computations. So, what are the different kinds of linear algebra operations that you see. So, the first category is this is called BLAS level 1; this is essentially vector or vector vector kind of operations like scaling a vector, finding the Euclidean norm or computing the dot product of 2 vectors.

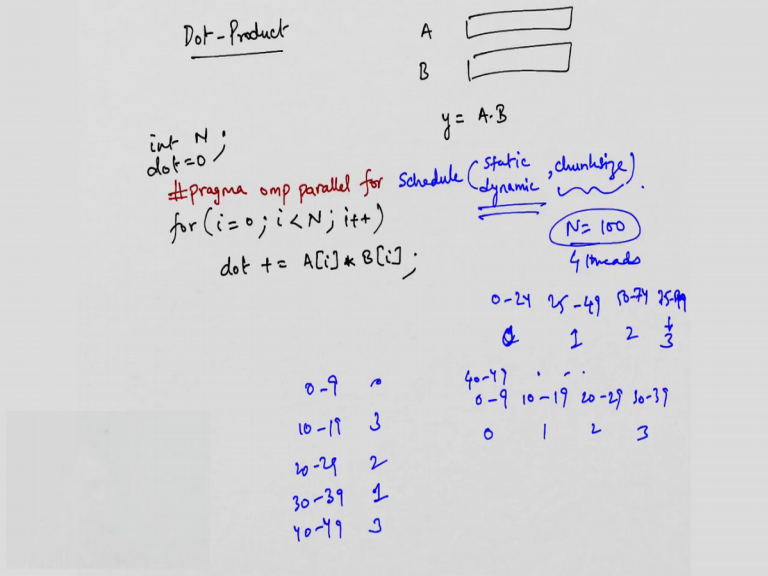
Then you have BLAS level 2 and this primarily constitutes of matrix vector operations. So, finding the product of a matrix and a vector and there are loads of other operations

like triangular solve and so on and the third category is matrix matrix operations and the simplest example of this is matrix multiply; you want to multiply two matrices and get the results and it is interesting that are lot of scientific computations can actually be performed using all the operations that are available in BLAS.

So, you can solve a system of linear equations using these routines and a whole lot more. At the end of the day, you are doing a lot of matrix or matrix vector kind of operations. Alright; so, let us start with some very simple BLAS 1 level routines. So, the simplest one is well let us I am not going to; obviously, go through all the routines that there are, but you can go to the net and find out more about these, but I will just cover a few as examples and show you how you can paralyze them.

So, let us start with dot product; so, this is something we have already seen.

(Refer Slide Time: 03:03)



So, what do we want to do we are given two vectors A and B and we want to compute the dot product. So, we want to compute y equal to A dot B; we have actually already done this. So, let us first write the sequential code; what will the sequential code look like? So, suppose you are given the size of the vectors in a variable N and now you compute the dot product as follows; simple loop for i is equal to 0; i is less than N; i plus plus and you simply say dot plus equal to A i times B i and initially you are going to initialize this dot to 0. And we are going to assume that all of these are floating point double precision floating point numbers. So, all of these are of type double in see.

So, unless specified otherwise we will just assume all data to be of type double. So, if I want to paralyze this; how do I do that? So, let me make some space; so, here is the simplest way I can do this using openmp. So, I just put a hash pragma here; so, you recall what this does. So, this is a combination of two different directives hash pragma omp parallel and hash pragma omp for; I have combined them together into a single directive.

So, what will this do? This will distribute this loop amongst the threads, whatever you set as the number of threads; it is going to launch that many threads and distribute this loop to the different threads but how does it do the distribution? So, you can actually specify how you want that to be done by using clause; it is called the schedule clause. So, you can say schedule static or dynamic and you can specify chunk size.

So, what static does is that; it basically divides the iterations statically amongst the threads; it is not figured out at runtime; it is done before that only at compile time only its fixed that which thread is going to do what work. For instance, if your arrays of size 100 and let us say that there are 4 threads; if N is equal to 100 and there are 4 threads and it is going to give 0 to 24 to thread number 0; 25 to 49 to thread number 1 and then 50 to 74 to thread number 2 and 75 to 100 to sorry 99 to thread number 3; it is going to do that statically.

Now, if you wanted to be done differently. So, let us say you want smaller chunks you want to give in units of let us say 10; then you can specify chunk size as 10. So, if you say schedule static comma 10; what it is going to do? It is going to give 0 to 9 to thread 0 into 19 to thread 1; 20 to 29 to thread 2, 30 to 39 to thread 3 and then back here 40 to 49 and so on; it will just distributed round robin alright.

But typically what you want is; you do not want to statically say that which thread is going to do what. Depending on the way the threads of scheduled, some threads may get freed up early and some threads may have a lot of work still remaining; depending on the order in which they are scheduled; that is not in your control. So, what you would ideally like to do is let the runtime figure out at runtime let it figure out that; which is the thread which is free; which are the thread is free let it pick up the next chunk of work; let me not statically assign the work with these threads.

So, then what I can do is I can just specify the schedule as dynamic. So, if I say schedule dynamic and if I use a reasonable chunk size; let us say chunk size of 10; if your

schedule is dynamic; it will give 0 to 9 to some thread whichever is free; it might be any thread; let us its thread 0, then 10 to 19 may go to thread 3; maybe that is it finds that thread 3 is free at that time and then 20 to 29 at finds thread 2 is 3 and then 30 to 39 to thread 1 and then 40 to 49 or maybe its thread 3 is the one which has freed up first; its computed its work. So, it is going to give it to thread 3.

So, it figures it out dynamically that whichever third is free; let me figure that out and give it some work. And you do not want to make the chunk size too small also; if you make it a chunk size of 1 or something and let us say that it is a very small loop which does not do a whole lot of work. If it is doing some considerable amount of work; it is taking sometime within each iteration; then it is to set a chunk size of 1.

But if let us say all you are doing is a multiplication of two numbers and you are setting a chunk size of 1; the overhead of you know giving the work allocating the work and so, on is going to be more than the operations being performed inside. So, you do not want to set a chunk size which is too small. So, if we want to compute the dot product; how would I ideally want to divide up the vectors would chunk size of 1 bigger? Would chunk size of hundred bigger?

Student: Cache size; according to the cache size.

According to the cache size; why is cache important in dot product? So, one of two things has to happen for cache to be effective. Either data has to be contiguous which you are accessing or you are reusing data. So, in this case are we reusing data? No. When you are reusing data; then cache size is very very important.

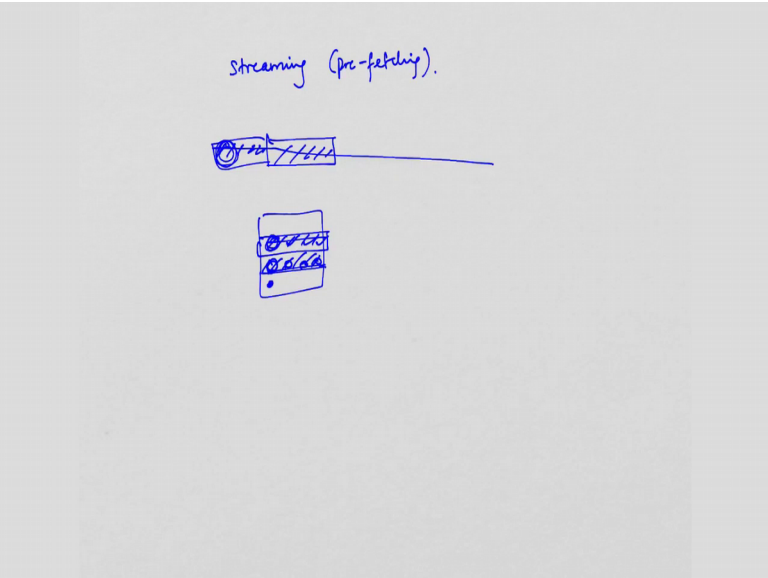
Because you want to fit as much data into the cache, so that you can reuse it and it still stays in the cache; it does not get out of the cache. For instance in matrix multiply if you multiply two very big blocks; then the data is bound to get out of the cache. Again dot product we are visiting each element only once; so, there is no reuse; is it that point clear?

So, I am not going to benefit by cache reuse, but the locality is going to come into play. So, once you load some data; it loads the entire cache line.

And then I am going to access the other elements of that cache line because I am accessing data contiguously. So, that is where the cache comes into play but for that the cache size is not important; the cache line size is important; what is the size of one cache line. Does it load 4 elements in a cache line or 8 elements in a cache line; that is important because that is the number of accesses that you will now find in the cache.

But the total cache size is not important; that is important more when you are doing reuse of data from the cache. So, there is another interesting thing that happens with caches; so, what modern day architectures support is something called streaming.

(Refer Slide Time: 10:24)



Or kind of pre fetching; if you are accessing data sequentially; which happen in the case of dot product or scaling a vector whenever you are accessing a vector; you are accessing data sequentially, so that is called streaming. You are working on a stream of data; so, if you are accessing data sequentially contiguously; then what it does is that it is able to figure that out; maybe you access the first element; it got it into the cache, but in got that entire cache line.

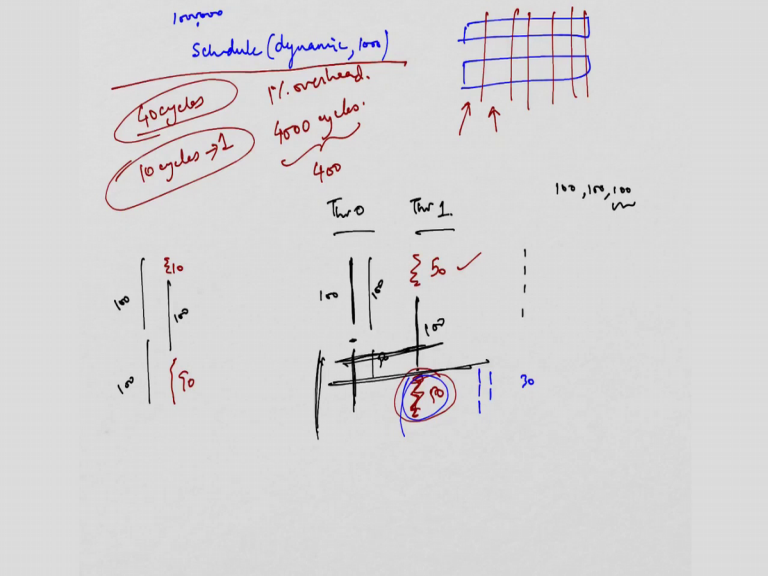
Now, you access this element, you access this element you access this element and then you again access the next sequential element. And at that time it picked up this entire data and put it in the cache line you access this element, this element, this element. So, now, it is able to figure out after about maybe you fetch two cache lines or something right of sequential data. The architecture is able to figure out that you are accessing data

sequentially; it its intelligent enough and now what it starts doing is without you accessing the next element; it will start automatically pre fetching it into the cache.

So, even that one element latency that you used to incur of when you used to run out the cache line and the next element you would have to make a memory access; you would incur a memory latency even that is gone. It is able to figure out that you are accessing data sequentially and it automatically pre fetches the data into the cache. So, that is something called streaming pre fetching which you should be aware of. So, it is always good to access data sequentially contiguously; you really benefit with a lot of architectural features by doing that.

So, how do we want to divide the work for a dot product; how do I want to divide the processing of these vectors; how do I want to give some chunk to each thread to work on?

(Refer Slide Time: 12:10)



Student: Equally in the all threads.

Equally to all the threads; so, what is the disadvantage of that? So, the only disadvantages as we discussed that sometimes the threads are not able to get scheduled due to some reason or the other; what could be the possible reasons. So, typically in all the operating systems that you there are demons; that wake up from time to time and do something maybe a clock interrupts maybe some file system server, networks, network

device server and so, on. So, these demons wake up from time to time and then if one of these demons wakes up; it your thread is just not getting scheduled.

So, you are kind of like in weight mode; so, maybe one of the threads does not get scheduled. So, if you are doing equal distribution of work and one thread gets stuck; then your entire dot product computation is waiting for that thread to come back to finish its work and come back; so, what is a good alternative to that?

Student: Whichever compiler (Refer Time: 13:14).

What will I say schedule dynamic?

And what should my chunk size be not one fourth of the total thing. So, if I have let us say a million entries that I want to compute the dot product over; 1000 or 10000 would be good enough. I do not want to make it too small; I do not want to make it too large. If I make it too small then the overheads of dynamic scheduling are going to play a bigger role and if I make it too large; then I may not get good load balancing because at the end; the thread which has the longer work, it may get stuck and may come back later alright.

So, maybe I do with 1000 that s not an unreasonable thing to do. So, essentially what am I doing; I am just dividing this vector into chunks and then as and when a processor becomes free, it comes and accesses.

Student: Is it better than dividing it equally in the dynamic scenario?

Dividing it equally in the dynamic scenario let us say we have 2 threads. So, let us say that each of these is a chunk of size 100 and I have data which is divided into 3 chunks of size 100. So, now what happens is that there are 300 iterations; I have divided into 3 chunks each of 100 iterations and now I want to schedule this on the processes. So, if I am using chunk sizes of 100; what happens in that case. So, let us say thread 0 starts running this and then thread 1 for some reason is not able to get scheduled for some time.

So, maybe about equivalent of doing 50 operations; half the time it is wasted, it is not able to schedule the thread and finally, it wakes up and it schedules the thread and then at this point of time; thread 0 becomes free and now it picks up that last piece of work and executes it and again this thread is sitting idle at this point in time not doing anything for another 50; let us say cycles. Let us assume that 1; it takes 1 cycle for 1 iteration; so, its

wasted 50 cycles here; which we could not do anything about, but this 50 cycles; it has wasted what would have happened if I had smaller chunks? If I had chunks of 10.

So, what would have happened is that roughly 10 of those would have got scheduled here. So, you would have done 100 work here and till this point how much work would you have done? So, 100 work would have been done here; 50 would have been done here, 100 would have been done here. So, what 250 units of work would have been done at this point in time and now how much work are you left with? Another 50 units, but since your chunk size is 10; both of them schedule it together and maybe this one schedules it longer.

So, in 30 units of time you are done whereas, in this case you know 50 units got wasted. So, this seems small, but you can construct examples where this becomes larger and larger. So, you could waste as much time as the size of the chunk; if this job ended over here. Let me construct another example; so, 100 units let us say that the first 10 units got free and then 100 units and then 100 units. So, what has happened? Around 90 units have gotten wasted.

So, smaller chunk size is always helped. So, with bigger chunk sizes there will always be one job; whichever one schedules get scheduled last, the others get up get free they have to wait for that last one to complete and that could be as much as the length of that job; the work, the amount of work that thread has to do the chunk size, but with smaller chunk sizes that gets load balanced, everybody can share that work. But again you do not want to make it too small because if you make it too small, then your overheads become significant.

Student: What do you mean by overheads (Refer Time: 17:15)?

So, it has to do some work; so, every time a chunk ends. So, what does the compiler do? The compilers going to substitute some code over there; which is going to check which threads are free, pick up a thread; allocated the work dynamically tell it that since you are free why do not you do this work; how is that done? It has to fill up some data structures in the memory; tell that thread that he pick up this information and start working.

So, all of that has to be done, so those are the overheads and that it has to do every time one chunk is completed because it has to dynamically figure out at that time that who am

I supposed to give the next work to. So, this is a processing involved over there; that is the overhead we are talking about

Student: Sir, is there any optimal way to select the chunk size?

Is there an optimal way to select the chunk size? Let us say that I am doing 10 cycles worth of work in one iteration and the overhead is about 40 cycles; I mean I am just arbitrarily cooking up numbers. So, this is the amount of work I am doing in 1 iteration. So, think of it is 10 nanoseconds and the overhead of doing the dynamic scheduling is 40 nanoseconds; then I may want to at least schedule about I want this to be 4000 cycles so that this becomes about 1 percent overhead; I do not want to overhead to be more than 1 percent.

So, I will select this to be about 4000 cycles; which means what? Which means about a chunk size of 400 like that is one way of doing it; you want to pick the smallest chunk size for which the overhead is not too large. Because smaller chunk size means that at the end you are going to get good looked balancing amongst the threads. So, you want to pick a smaller chunk size.

So, that was dot product; so, the crux of the story is that. So, we divided it up into lots of small small pieces maybe 100 elements or something each; I say schedule dynamic. So, as and when a thread becomes freed picks up 1 chunk and goes and end of it and because there is at least a 100 elements or let us say 1000 elements; it benefits out of the data locality, whatever is floating in the cache and also the streaming comes into play; so, it is able to get good performance.