Chapter 5

Basics of Parallelization

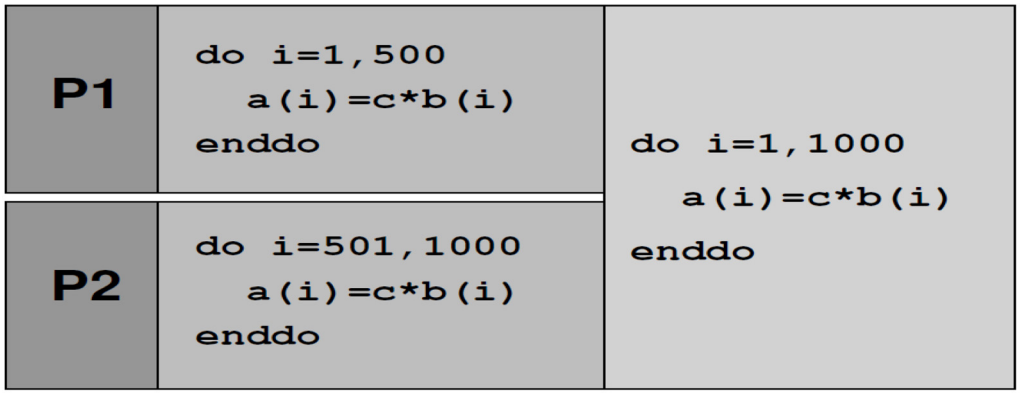
Why parallelize?

Two reasons:

- A single core would be too slow
- The amount of memory a single system does not suffice
  - The alternative is to use out-of-core techniques.

Parallelization methods

- **Data parallelism**. Apply "same" operation to all elements of an aggregate (array, set, nodes in a graph, ...)
  - Fine-Grained parallelism. Vector machines. Microprocessor vector extensions.
  - Medium-grained parallelism. "A few" elements per processor.

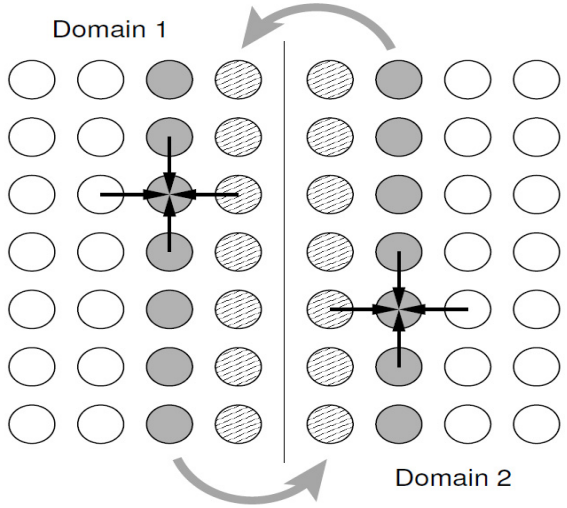


  - Coarse-grained parallelism. Numerous element per processor.

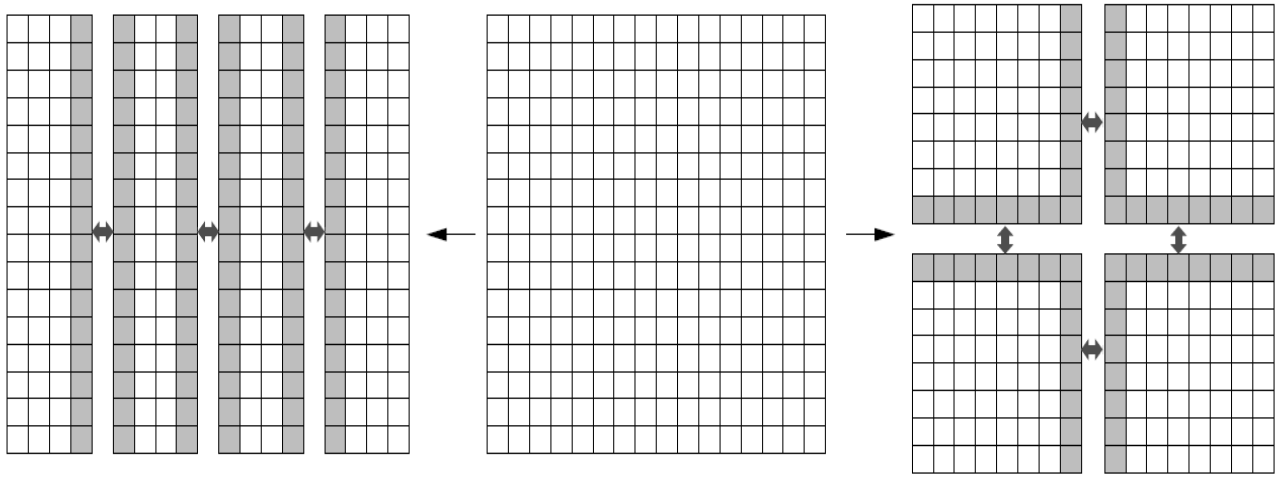
Example: consider the Jacobi algorithm from chapter 3

```
1  double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
2  integer :: t0,t1
3  t0 = 0 ; t1 = 1
4  do it = 1,itmax     ! choose suitable number of sweeps
5    do k = 1,kmax
6      do i = 1,imax
7         ! four flops, one store, four loads
8         phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
9                        + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
10     enddo
11   enddo
12   ! swap arrays
13   i = t0 ; t0=t1 ; t1=i
14 enddo
```

The way to implement it in parallel for a distributed memory machine is to use ghost layers for communication:



There are multiple ways to partition the data across processors (domain decomposition):



Which one is better ?

Vector notation for data parallel algorithms

Next, we study several algorithms where parallelism can be easily expressed in terms of array operations. We will use Fortran 90 to represent these algorithms.

Simplistic timing figures will be given in some cases for SIMD machines.

In these timings, subscript computations and memory access/ communications costs will be ignored.

Time to execute a vector operation

Let us start with the simplest possible situation. Consider the following generic vector operation:

```
a(1:n) # b(1:n)
```

Consider an array machine with *P* arithmetic units.

The execution time is:

$$t_{parallel} = \left\lceil \frac{n}{P} \right\rceil t_{\#}$$

where $t_{\#}$ is the time to execute one # operation.

Reductions in Fortran 90

A typical reduction is `sum(array)` which returns,as we should expect, the sum of the elements of an integer, real, or complex array. It returns zero if `array` has size zero.

Others include:

`all(mask)`	Returns the logical value .true. if all elements of of the logical array `mask` are `.true.` or `mask` has size zero, and otherwise returns the value `.false.`
`any(mask)`	Returns the logical value `.true.` if any of the elements of the logical array `mask` is `.true.` , and returns the value `.false.` if no elements are `.true.` or if `mask` has size zero.
`count(mask)`	Returns the number of `.true.` values in `mask`.
`maxval(array)`	Returns the maximum value of the elements of an integer or real array.

`minval(array)`	Returns the minimum value of the elements of an integer or real array.
`product(array)`	returns the product of the elements of an integer, real, or complex array. It returns 1 if `array` has size zero.

All these functions have an optional argument `dim` if this is present, the operation is applied to all rank-one sections that span right through dimension `dim` to produce an array of rank reduced by one and extends equal to the extents in the other dimensions. For example, if `a` is a real array of shape [4,5,6], `sum(a,dim=2)` is a real array of shape [4,6] and element (i,j) has value `sum(a(i,:,j))`.

The functions `maxval`, `minval`, `product`, and `sum` have a third optional argument, `mask`. If this is present, it must have the same shape as the first argument and the operation is applied to the elements corresponding to true elements of `mask`; for esample, `sum(a,mask=a>0)` sums the positive elements of the array `a`.

Two other useful Fortran 90 functions.

`1. spread(source,dim,ncopies)`

Returns an array of rhe same type as `source` but with rank increased by one over `source`. `Source` may be a scalar or an array. `Dim` and `ncopies` are integer scalars. The result contains `max(ncopies,0)` copies of source, and element $(r_1,\ldots,r_{n+1})$ of the result is source $(s_1,\ldots,s_n)$ where $(s_1,\ldots,s_n)$ is $(r_1,\ldots,r_{n+1})$ with subscript `dim` omitted (or `source` itself if it is a scalar).

Example of use:

```
a=spread(x,dim=2,ncopies=n)+spread(x,1,n)
w=sum(abs(a),dim=1)
```

is equivalent to:

```
do i=1,n
   w(i)=0
   do j=1,n
      w(i)=w(i)+abs(x(i)+x(j))
   end do
end do
```

`2. maxloc(array[,mask])`

Returns a rank-one integer array of size equal to the rank of `array`. Its value is the subscript of an element of maximum value.

Time to Execute a Reduction

Consider a reduction such as:

```
r = sum(a(1:n)) = a(1) + a(2) + a(3) + ... a(n)
```

or, in general

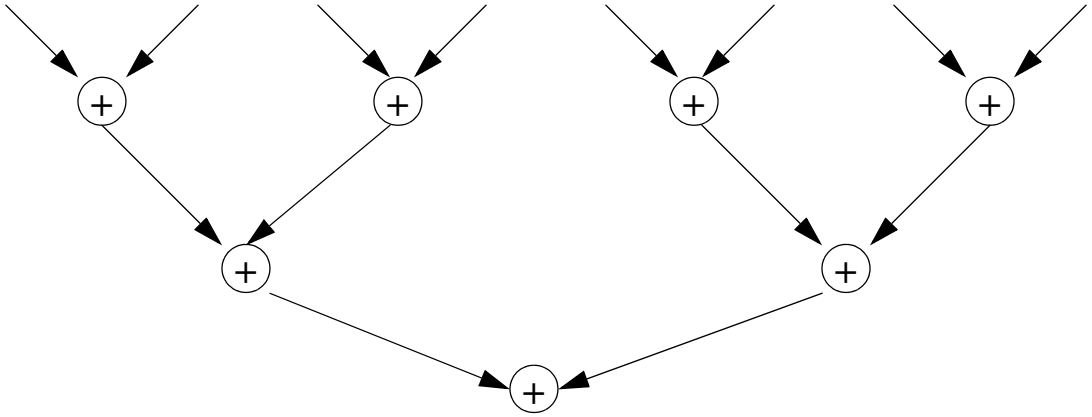
```
r =  a(1) # a(2) # a(3) # ... a(n)
```

A sequence of $\lceil \log_2 n \rceil$ vector operations of length n/2, n/4, ..., 1 suffices to compute the reduction (assuming associativity of the # operation).

In the case of an array machine, there are two cases. First, if $P < n/2$, and if we follow the approach presented in our discussion of reductions in OpenMP, we have:

$$t_{parallel} = \left(\left\lceil \frac{n}{P} \right\rceil - 1\right) t_{+} + (P-1) t_{+}$$

If the final reduction can also be done in logarithmic time using a reduction tree approach:



In this case, the execution time is:

$$t_{parallel} = \left(\left\lceil \frac{n}{P} \right\rceil - 1\right) t_{+} + \lceil logP \rceil t_{+}$$

If *P* >= *n/2*, the time is:

$$t_{parallel} = \lceil logn \rceil t_{+}$$

The # operation could be a simple arithmetic operation such as s + or * or it could be a more complex binary operation. For example, to implement `maxloc` in logarithmic time we could define an operation on two pairs consisting of a value and a location:

```
(v1,loc1) # (v2,loc2)=
        if v1 < v2 then return(v1,loc1)
                   else return(v2,loc2)
```

And, to implement an in logarithmic time an operation that finds the location of the first negative value in a vector we could define the following similar operation:

```
(v1,loc1) # (v2,loc2)=
        if v1 < 0  then return(v1,loc1)
                   else return(v2,loc2)
```

Notice that both of these operations are associative (but NOT commutative).

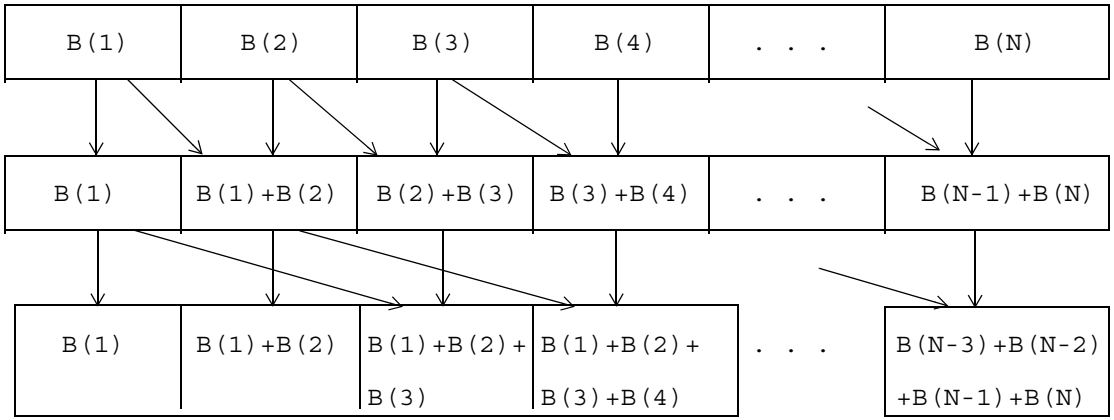
Parallel Prefix

Consider the following loop:

```
A(0)=0
DO  I=1,N
    A(I)=A(I-1)+B(I)
END  DO
```

The loop seems sequential because each iteration needs information on the value computed in the preceding iteration.

However, we can use a *parallel prefix* approach to compute the value of vector A in parallel as follows:



A parallel program implementing this strategy under the assumption that $N=2^k$ is:

```
A(1:N)=B(1:N)
DO I=0,K-1
   A(2**I+1:N)=A(2**I+1:N)+A(1:N-2**I+1)
END DO
```

For an *array machine* with the number of procesing units *P>=n-1:*

$$t_{parallel} = t_{+}\lceil logn \rceil$$

As in the case of reduction, parallel prefix can be applied to any associative binary operation.

Matrix-Vector Multiplication

In mathematical notation:

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \cdots \\ V_n \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} A_{1i} V_i \\ \sum_{i=1}^{n} A_{2i} V_i \\ \cdots \\ \sum_{i=1}^{n} A_{mi} V_i \end{bmatrix}$$

In Fortran:

```
do i=1,m
   R(i) = 0
   do j=1,n
      R(i) = R(i) + A(i,j) * V(j)
   end do
end do
```

The inner loop performs a dot product (or inner product) of two vectors. It can be represetned in Fortran 90 as follows:

```
do i=1,m
   R(i)=DOT_PRODUCT(A(i,1:n),V(1:n))
end do
```

The dot product is a vector multiplication (of length *n,* in this case) followed by a reduction.

In an array machine or in a multiprocessor, the time if *P*>*n* is:

$$(m(\lceil log n \rceil t_{+} + t_{*}))$$

Alternatively, by interchanging the loop headers, the program could be written as follows:

```
do j=1,n
   do i=1,m
      R(i) = R(i) + A(i,j) * V(j)
   end do
end do
```

This leads to the following sequence of vector operations:

```
do j=1,n
   R(1:m)=R(1:m)+A(1:m,j)*V(j)
end do
```

In an array machine or in a multiprocessor, the time (if P > m) is:

$$(t_+ + t_*)n$$

Matrix-Matrix Multiplication

1. *Inner product method.*

   Matrix multiplication is usually written:

```
do i=1,n
   do j=1,n
      do k=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end do
   end do
end do
```

   The most direct translation of this program into vector form is:

```
do i=1,n
   do j=1,n
      C(i,j)=DOT_PRODUCT(A(i,1:n),B(1:n,j))
   end do
end do
```

The time on an array machine or multiprocessor if $P > n$ is:

$$(t_+\lceil \log n \rceil + t_*)n^2$$

2. *Middle-product method* (n-parallelism)

This is obtained by interchanging the headers in the original matrix multiplication loop.

```
do j=1,n
   do k=1,n
      do i=1,n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end do
   end do
end do
```

The direct translation of this loop into vector form is:

```
do j=1,n
   do k=1,n
      C(1:n,j)=C(1:n,j)+A(1:n,k)*B(k,j)
   end do
end do
```

Alternatively, the headers could have been exchanged in a different order to obtain the loop:

```
do i=1,n
   do k=1,n
      C(i,1:n)=C(i,1:n)+A(i,k)*B(k,1:n)
   end do
end do
```

The time in an array machine is:

$$(t_+ + t_*)n^2$$

Sorting in Fortran 90.

There are many parallel sorting algorithms. We will discuss two very simple ones in this chapter and more elaborate algorithms later in the semester.

Perhaps the simplest sorting algorithm is *bubble sort*. (Text extracterd from Kumar et al. *Introduction to Parallel Computing*) It compares and exchages adjacent elements in the sequence to be sorted. Given the sequence $a_1, a_2, \ldots, a_n$, the algorith first performs n-1 compare-exchange operations in the following order: $(a_1,a_2),(a_2,a_3), \ldots(a_{n-1},a_n)$. This step moves the largest element to the end of the sequence. The last ement in the sequence is then ignored, and the sequence of compare exchanges is applied to the resulting sequence. The sequence is sorted after *n-1* iterations. The algorithm is as follows:

```
do i=n-1,1,-1
   do j=1,i
      if (a(j) > a(j+1) ) swap (a(j),a(j+1))
   end do
end do
```

Where *swap(a,b)* is just the sequence

```
t=a
a=b
b=t
```

This algorithm can be easily parallelized as discussed later on.

For vectorization, we will use the following slightly modifed version known as *odd-even transposition*:

```
do i=1,n
   if i is odd then
      do j=0,n/2-1
         if (a(2*j+1)>a(2*j+2)) swap(a(2*j+1),a(2*j+2))
      end do
   end if
   if i is even then
      do j=1,n/2-1
         if (a(2*j)>a(2*j+1)) swap(a(2*j),a(2*j+1))
      end do
   end if
end do
```

The algorithm alternates between two phases: odd and even. During the odd phase, elements with odd indices are

compared with their right neighbors, and if they are out of sequence they are exchanged. Similarly, during the even phase, elements with even indices are compared with their right neighbors, and if they are out of sequence they are exchanged.

Vectorization is quite simple:

```
do i=1,n
   if i is odd then
      where (a(1:n-1:2)>a(2:n:2))
         swap (a(1:n-1:2),a(2:n:2))
      end where
   end if
   if i is even then
      where (a(2:n-2:2)>a(3:n-1:2))
         swap (a(2:n-2:2),a(3:n-1:2))
      end where
   end if
end do
```

Bubble sort is not a very efficient algorithm. It takes *n(n-1)/2* comparisons to complete. The parallel version reduces that to *n* steps, but a good sequential algorithm only requires a number of comparisons proportional to *n log n*. And there are paralle algorithm that require time proportional to $\log^2 n$. So this is ok, but not great.

A better sorting algorithms in some situations is *radix sort*. This was the algorithm used to sort punched cards with electro-mechanical devices.

The idea is that the values to be sorted are assumed to be numbers in a certain radix. Integers could be radix 10 or 2 depending on the circumstances. For punched cards, it was base 10. In today's machines, we could assume base two, but any other base can be assumed. When values are names, base 26 can be assumed.

Radix sort, goes through all the "digits" starting with the less significant one. For each digit it processes the whole sequence. Elements of the sequence are placed in separate buckets, one for each possible digit. Placement in the buckets is in the order theelements appear in the sequence. After processing all elements for a particular position, the buckets are catenated to create the sequence for the next position.

Consider for example the following sequence:

223, 148, 221, 071, 138, 131.

After the first step, the sequence will be separated as follows:

bucket	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
		221		*223*					*148*	
		071							*138*	
		131								

After catenation, we get: *221,071,131,223,148,138*.

Now, the digits in the second position are processed:

bucket	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
			221	*131*	*148*			*071*		
			223	*138*						

Again,the buckets are catenated: *221,223,131,138, 148,071.*

Then, the digits in the third position are processed:

bucket	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	071	*131*	*221*							
		138	*223*							
		148								

Finally, the sorted sequence is obtained by catenating the buckets: *071,131,138,148, 221, 223.*

The algorithm can be easily implemented in Fortran 90 using the `pack` intrinsic function. `Pack(array, mask)` returns a one-dimensional array containing the elements of `array` to pass the `mask`.

Thus, assuming that the sequence to be sorted is in vector `a`, and that the elements are in base *b* and contain *d* digits each, we can proceed as follows:

```
do i=1,d
   m_0(1:n) = the digit in a(1:n) with weight b^(i-1) is 0
   m_1(1:n) = the digit in a(1:n) with weight b^(i-1) is 1
   ...
   m_b1(1:n) = the digit in a(1:n) with weight b^(i-1) is b-1
   a=(/pack(a,m_0),pack(a,m_1),...,pack(a,m_b1)/)
end do
```

In particular for base 2, only one mask is needed:

```
do i=1,d
   m=mod(a,2**i)<2**(i-1)
   a= (/pack(a,m),pack(a,.not.m)/)
end do
```

Pack can be implemented in parallel using the primitives discussed earlier in class:

```
function pack(a,m)
   order=parallel_prefix(m)
   where (m)
      temp(order(:))=a(:)
   end where
   pack=temp
   return
end
```