22nd Summer School on PARALLEL COMPUTING

Linear algebra algorithms

Fabio Affinito – f.affinito@cineca.it

SCAI dept - CINECA

CINECA

Introduction

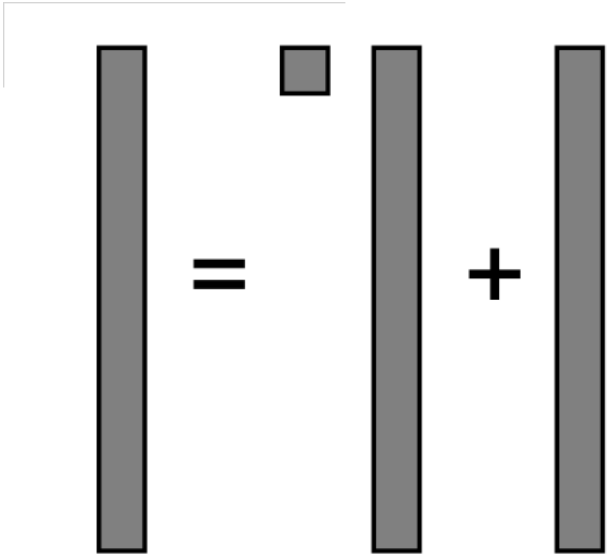
We will study the serial and parallel approach to the solution of simple operations such as:

- Vector axpy: $z = a\,x + y$
- Vector dot product $a = x \cdot y = x^T y$
- Matrix-vector product $y = A\,x$
- Linear system solution: solve $A\,x = y$ for x

Serial axpy

To compute the linear combination

z = a x + y



```
for i = 1:n
    z(i)= a*x(i)+y(i)
end
```

- order n operations
- easy optimize to vector or hierarchical memory architectures
- all operations affect independent components of the vector output

Serial dot product

To compute the dot-product $a=x^T y$

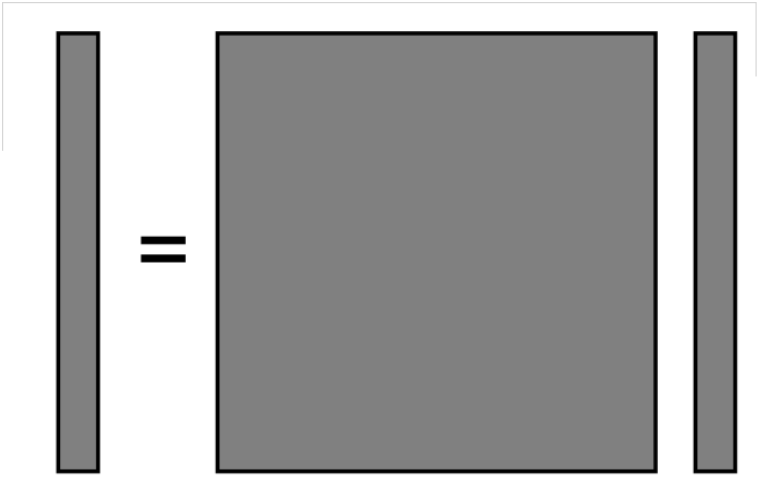
```
a=0.0
for i=1:n
    a=a+x(i)*y(i)
end
```

- order n operations
- easily optimizable to vector or hierarchical memory architectures
- all operations affect the single scalar output a

Serial matrix-vector product

To compute the matrix vector product y = A x

```
for i = 1:n
     y(i) = 0.0
     for j = 1:n
          y(i)=y(i)+A(i,j)*x(j)
     end
end
```



- order n^2 operations
- Easily optimizable for vector or hierarchical memory architectures
- All components of x affect all components of y via non-zero structure of A
- For sparse matrices, many of these operations can be eliminated resulting in O(n) operations

Solving linear systems

To solve the linear system Ax=y for given x, we have a large number of choices. These depend on a variety of factors:

- the system size, n
- the sparsity of the matrix A
- the nonzero structure of a sparse matrix
- the type of problem giving rise to the linear system

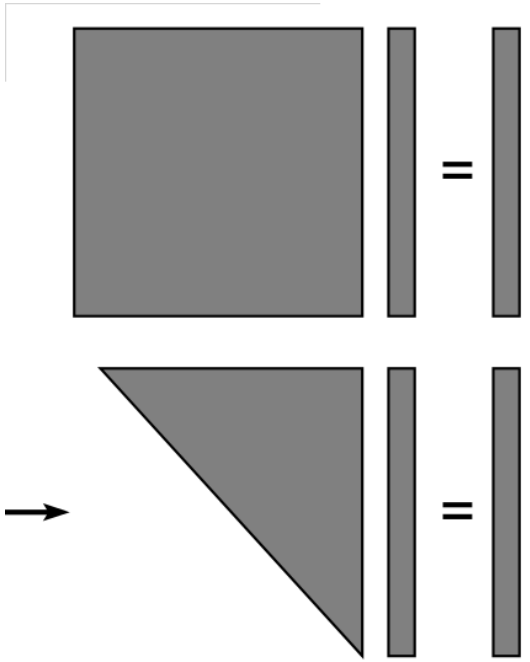
We will deal with these very broadly investigating two methods in detail: a direct solver (Gaussian elimination) and an iterative solver (Jacobi), to help illustrate how one might choose between various methods for different problems in serial and parallel.

For both we will assume that the solution x to the system exists and that the problem is not ill-conditioned

Serial gaussian elimination

Idea: solve Ax=y using diagonal values and elementary row operations to eliminate the lower triangular portion of A. Then use back-substitution to obtain x.

```
for k=1:n-1
     for i=k+1:n
          for j=k+1:n
               A(i,j)=A(i,j)-A(k,j)*A(i,k)/A(k,k)
          end
     y(i)=y(i)-y(k)*A(i,k)/A(k,k)
end
for i=n:-1:1
     x(i)=y(i)/A(i,i)
     for j=i+1:n
          x(i)=x(i)-A(i,j)*x(j)/A(i,i)
     end
end
```

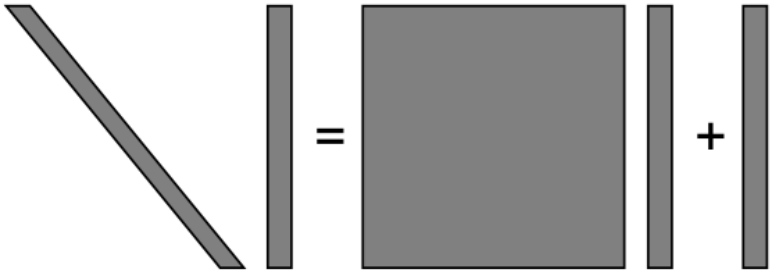


- order n^3 operations
- guaranteed to work for nonsingular A changes)

Serial Jacobi iteration

Idea: solve the system Ax=y by decomposing A = D + LU

$x^{k+1} = D^{-1}(y - LU\, x^k)$



Given x^0 and d>0, set $x=x^0$, w=Ax

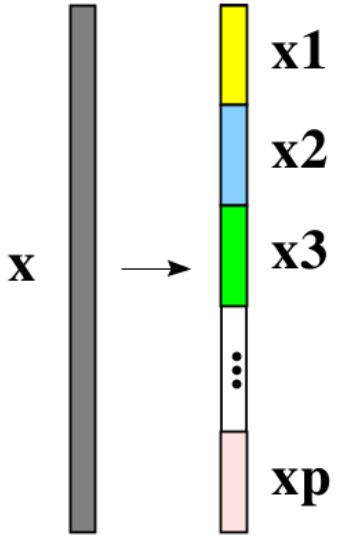
```
while |w-y|>d
    for j=1:n
        x(j)=x(j)+(y(j)-w(j))/A(j,j)
    end
    w=Ax
end
```

- each iteration requires O(n) operations plus one matrix-vector product
- only converges for diagonally dominant matrices A

Let's go parallel

Now consider the vectors x as decomposed among p processors.

One must therefore determine how the vectors are decomposed among processors.



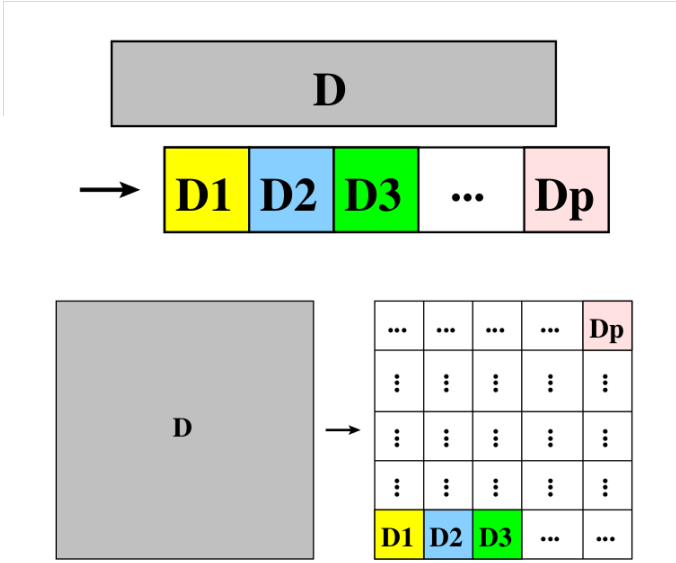
Choices include:

- a decomposition based on the problem that gives rise to the linear system, such as a PDE or other domain decomposed simulation, with a previously assumed parallel decomposition
- a decomposition that is customized to solve the given linear system in an optimal manner for a given solver algorithm

Subdomain based decomposition

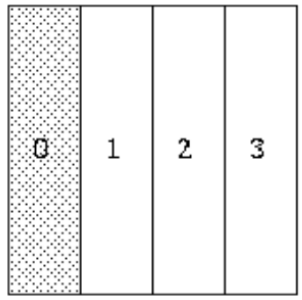
Domain decomposition parallelism sub-divides a problem among processors into contiguous sections, in an effort to minimize the ratio of surface area to volume of data on the processor. The linear algebra is then performed on the resulting decomposition.

In a 1D decomposition, the data on each processor is neighbored by data on at most 2 other processors.
In a 2D decomposition, the domain is similarly split into a 2-dim processor grid

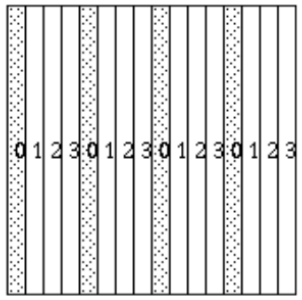


Solver based decomposition

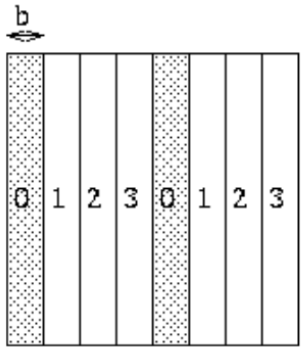
Dense linear algebra packages typically require a block cyclic data distribution, in an attempt to optimize the load balance and communication characteristics of the algorithms.



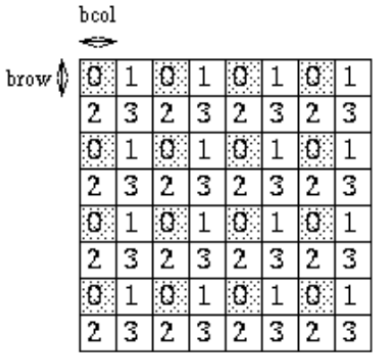
column blocked layout



column cyclic layout



column block cyclic layout



row and column block cyclic layout

Notation

For the following algorithms, we will assume a 1D decomposition. The global domain is mapped onto p processors.

We assume that each processor contains m = n/p values.

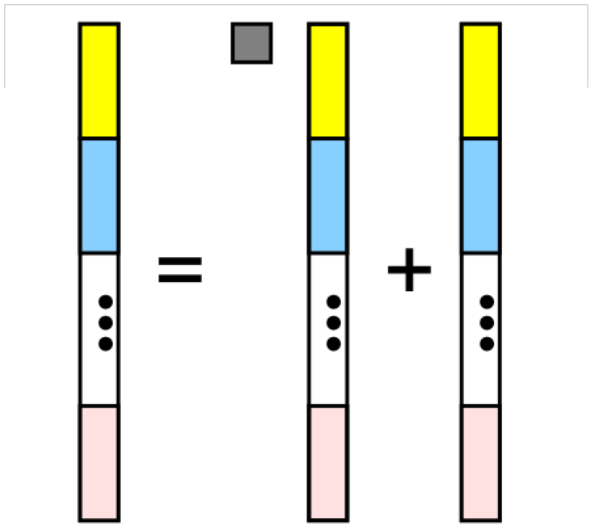
We will use q as the processor index. q=1,..,p

Parallel axpy

Since all operations in the linear combination z=ax+y affect only processor local components, the parallelization is trivial.

On each processor q containing local vector components x_q,y_q,z_q we independently perform:

```
for i=1:m
    z_q(i)=a*x_q(i)+y_q(i)
end
```



- No communication between processors is required
- Ideal strong scaling and weak scaling, as problem size and processor counts increase

Parallel dot-product

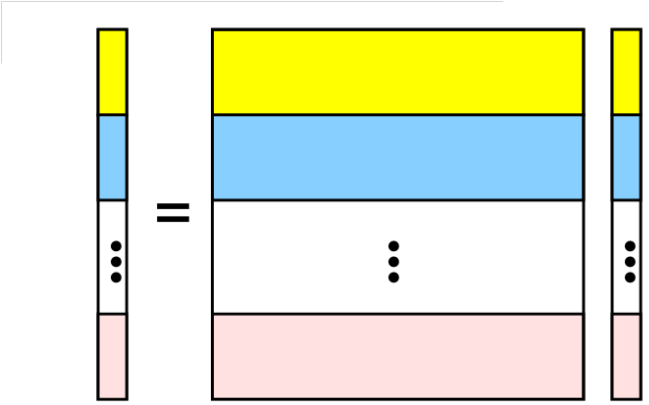
Nearly all operations in computing the dot-product $a=x^T y$ are processor local. Only the final scalar product requires communication between all the processors:

```
a_q=0.0
for i=1:m
    a_q=a_q+x_q(i)*y_q(i)
end
MPI_Allreduce(&a_q, &a ....MPI_SUM...)
```

Parallel matrix-vector product method 1

Local output y_q requires knowledge of global x:

```
MPI_Allgather(&x_q,m,MPI_DOUBLE,&x,n..)
for i=1:m
    y_q(i)=0.0
    for j=1:n
        y_q(i)=y_q(i)+A(i,j)*x(j)
    end
end
```



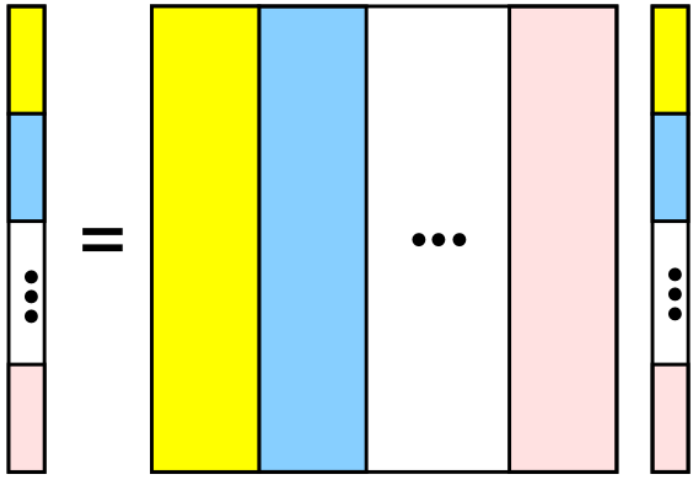
- if the matrix is sparse or it has special structure than we may use such properties appropriately

Parallel matrix-vector product method 2

Each processor computes a local contribution to the local output followed by reduction onto the appropriate processor:

```
for i=1:n
    y(i)=0.0
    for j=1:m
        y(i)=y(i)+A(i,j)*x_q(j)
    end
end

for q=1:p
    MPI_Reduce(&y,&y_q,m...)
END
```



Solving linear systems in parallel

Difficulties encountered when dealing with matrix-vector multiplication are similar to ones related to linear systems:

- Any solver algorithm must allow the solution to depend on the global rhs through the nonzero structure of the matrix
- There cannot exist simple solver algorithms that do not involve significant parallel communication
- We will consider only one example: the parallel Jacobi iteration

Parallel Jacobi iteration

Idea: build out of parallel vector operations and matrix-vector product.

Given x_q^0 and d>0

```
x_q=x_q^0
w = matvec(A,x)
r = sqrt(dot_product(w-y))
while (r>d)
    for j=1:m
        x_q(j)=x_q(j)+(y_q(j)-w_q(j))/d_q(j)
    end
    w = matvec(A,x)
    r = sqrt(dot_product(w-y))
end
```

- All the parallel communications are contained in the dot_product and matvec operations
- All other operations may run independently on each processor

Parallel linear algebra software

There are two kinds of parallel linear algebra algorithms used in available software: direct and iterative.

Direct solvers may be difficult in parallel at large scale. However optimized solver libraries do exist but their scalability is limited to some hundred processors.

- Dense approaches are modeled in LAPACK and BLAS and, for parallel machines in ScaLAPACK and PLAPACK
- Sparse matrices are treated in SuperLU, Amesos

Parallel linear algebra software (iterative)

- want convergence in few iterations, with a formula for constructing the sequence that is both scalable with problem size and processor number
- may be independent of matrix structure and sparsity, or may require system arising from certain kinds of problems
- some methods can scale up to 100k processors
- depending on the method the iterations may or may not converge for all A
- a number of packages provide iterative solvers along with interfaces to approximate direct methods to help accelerate convergence
- examples are: PetSC, HYPRE, ML

References

- This lecture is inspired by the course given by D.R. Reynolds @UCSD

- Two complete textbooks are:
  - B.N. Datta, Numerical linear algebra and applications
  - Golub, Van Loan, Matrix Computations