FORTRAN and MPI

Message Passing Interface (MPI)

Day 3

Course plan:

- MPI - General concepts
- Communications in MPI
  – Point-to-point communications
  – Collective communications
- Parallel debugging
- Advanced MPI: user-defined data types, functions
  – Linear Algebra operations
- Advanced MPI: communicators, virtual topologies
  – Parallel sort algorithms
- Parallel performance. Summary. Tendencies

Memory organization for multiprocessor systems

Computer memories are known to be never big enough for what you want to do.

The hardship of the unanimous programmer. (Year unknown.)

Two major solutions have been offered:

- a shared memory model, combined with the so-called interleaved storage (Cray-1, CDC Cyber 205, Fujitsu, NEC SX, CG Power Challenge, Tera computers etc.),
- distributed memory model (CM-2/200, Intel Paragon, Intel iPSC/860, MasPar, nCUBE2, Cray T3D/T3E etc.)

All memory systems are, in addition, hierarchical.

Each memory system can be viewed as a complex of several types of memories with different capacity, response time and price.

Memory organization

Computer memories are characterized by

- capacity (size) — Measured in bytes (B). Ranges from several tens of B for registers up to hundreds of GB for disks.
- functional design — hierarchy, protection
- response time — *access time* and memory cycle time
  - access time — also called *latency* - is the time needed for the memory to respond to a read or write request
  - memory cycle time — is the minimum period between two successive requests to the memory
  - memory bandwidth — the rate a which data from/to memory can be transferred to/from CPU.

Re: *memory cycle time*

For many reasons, if a memory has say, 80 ns response time, it can not be accessed every 80 ns.

Furthermore, the cycle time can be longer than the access time.

The latter can, for example, be a consequence of the so-called *destructive read*, which means that after reading, the contents of the memory cell is destroyed and must be recovered afterwards, which causes time.

Re: *memory latency* - the most difficult issue

The access time per word varies from 50 ns for the chips in today's personal computers to 10 ns or even less for cache memories.
It includes time to

- select the right memory chip (among several hundreds) but also
- time spent waiting for the bus to finish a previous transaction before the memory request is initialized.

Only after that the contents of the memory can be sent along the bus (or via some other interconnection) to registers.

The Latency Problem

The larger the latency, the slower the memory is.
For certain RAM chips, called Dynamic RAMs, with a latency of about 50 ns, we could have

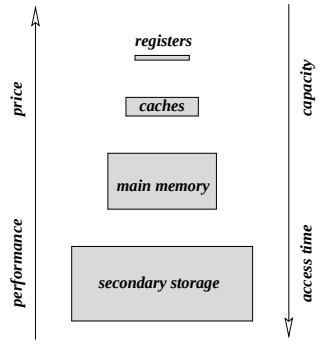
$$\frac{1\,\text{access}}{50\,\text{nanoseconds}} = 20\,\text{million accesses per second}$$

But typical desktop computers have a memory bus speed of 100 MHz, or 100 million memory accesses per second. How can we resolve this disparity between the memory latency and the bus speed?

A bus is simply a circuit that connects one part of the motherboard to another. The more data a bus can handle at one time, the faster it allows information to travel. The speed of the bus, measured in megahertz (MHz), refers to how much data can move across the bus simultaneously.

Bus speeds can range from 66 MHz to over 800 MHz and can dramatically affect a computer's performance.

The above characteristics have slightly changed but the principle remains.



The memory, closest to the processor registers, is known as *cache*. It was introduced in 1968 by IBM for the IBM System 360, Model 85. Caches are intended to contain the most recently used blocks of the main memory following the principle "The more frequently data are addressed, the faster the access should be".

Keywords:

- *cache lines*
- *cache hit*
- *cache miss*
- *replacement strategy* FIFO, LRU, ...
- *hit rate* is the percentage of requests that result in hits and depends on the memory organization and the replacement strategy
- *cache coherency* not considered a severe issue for distributed memory computers

Non-Uniform Memory Architecture - NUMA
cache-coherent NUMA - ccNUMA

Computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor.

NEC-SX-5: Multi Node NUMA Memory

The main memory configuration of SX-5M Multi Node systems includes both shared and NUMA architecture. Each node has full performance access to its entire local memory, and consistent but reduced performance access to the memory of all other nodes.

Access to other nodes is performed through the IXS Internode Crossbar Switch, which provides page translation tables across nodes, synchronization registers, and enables global data movement instructions as well as numerous cross node instructions. Memory addresses include the node number (as do CPU and IOP identifiers).

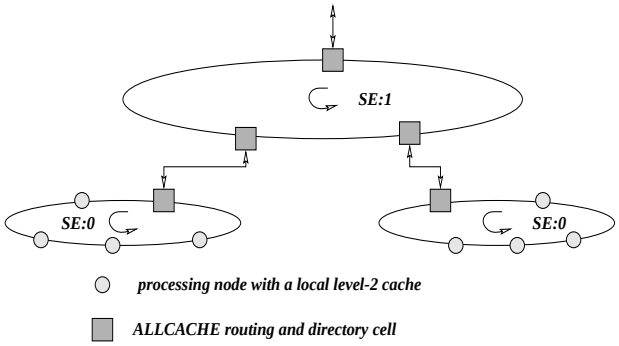
Latencies for internode NUMA level memory access are less than most workstation technology NUMA implementations, and the 8 gigabyte per second bandwidth of just a single IXS channel exceeds the entire memory bandwidth of SMP class systems. Even so, because the internode startup latencies are greater than local memory accesses, internode NUMA access is best utilized by block transfers of data rather than by the transfer of single data elements.

This architecture, introduced with the SX-4 Series, has been popularized by many products and lends itself to a combination of traditional parallel vector processing (microtasking and macrotasking) combined with message passing (MPI). Message passing alone is also highly efficient on the architecture.

ALLCACHE KSR1: good ideas which did not survive

The system hardware which assures the user to view the distributed memories, local to each processor, as a global shared address space, is the patented ALLCACHE memory management system. It provides the programmer with a uniform 2^{64} byte address space for instruction and data, called the *System Virtual Address space* (SVA). The contents of SVA locations reside in physically distributed memories, called *local caches*, each with a capacity of $2^{25} = 32$ MB. These local caches are second level caches, to distinguish from the first-level caches, which are the standard instruction and data caches. The data is stored in pages and subpages. Each local cache has $2^{11} = 2048$ pages, each of 128 subpages of $2^7 = 128$B. The unit of data which is exchanged, is a subpage.

The consistency is automatically maintained by taking into account the type of memory reference made by the processors - only read or modify. If the data is only read, the processor, which has initiated a memory request, will receive a copy of it and its address. If a modification is going to take place, the processor will receive the only instance of an address and its data.



The memory management is done by the so-called ALLCACHE Engines. The engines form a hierarchy of a potentially unlimited number of levels. The maximal length of the path to fulfill any memory request is $O(\log p)$, where p is the number of processors. One can view the hardware architecture as a fat tree, consisting of rings (called $SE:0$, $SE:1$ etc.), along which the search for a copy of a requested data proceeds. Each $SE:0$ ring connects 32 PEs. If the data is not found in the local $SE:0$ ring, to which the processor that issued the request belongs, the request is sent to the upper ring $SE:1$ etc. Each level has a growing bandwidth: 1 GB/sec for $SE:0$, 1, 2 or 4 for $SE:1$ and so on.

System parameters:

Model	BlueGene/L	BlueGene/P
Clock cycle	700 MHz	850 MHz
Theor. peak performance		
Per Proc. (64-bits)	2.8 Gflop/s	3.4 Gflop/s
Maximal	367/183.5 Tflop/s	1.5/3 Pflop/s
Main memory		
Memory/card	512 MB	2 GB
Memory/maximal	16 TB	442 TB
No. of processors	265,536	4221,184
Communication bandwidth		
Point-to-point (3-D Torus)	175 MB/s	350 MB/s
Point-to-point (Tree network)	350 MB/s	700 MB/s

The BlueGene/L possesses no less than 5 networks,
2 of which are of interest for inter-processor communication: a 3-D torus network and a tree network.

- The torus network is used for most general communication patterns.
- The tree network is used for often occurring collective communication patterns like broadcasting, reduction operations, etc. The hardware bandwidth of the tree network is twice that of the torus: 350 MB/s against 175 MB/s per link.

Cache-aware algorithms

Block-versions of various algorithms, combined with a proper data structure.

MPI: advanced features
User-defined (derived) datatypes

Grouping data for communications

Re: count and data-type parameters in MPI_Send and MPI_Recv

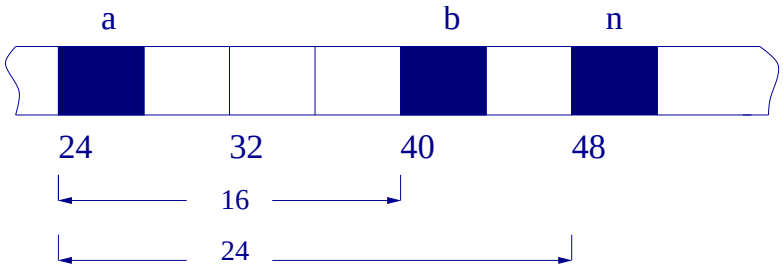
Imagine we have to send three variables from one PE to another, which are as follows:

Variable	Address	Value	Type
a	24	0.0	float
b	40	1.0	float
n	48	1024	int

The communication still can take place if we provide not all addresses but

- the address of a
- the relative displacement of b and n

Displacement of b: $40 - 24 = 16$
Displacement of n: $48 - 24 = 24$



Provide now the following information to the communication system:

- There are three elements to be transferred
- The first is `float`
- The second is `float`
- The third is `int`
- In order to find them ...
  - the first is displaced 0 bytes from the beginning of the message
  - the second is displaced 16 bytes from the beginning of the message
  - the third is displaced 24 bytes from the beginning of the message
- The beginning of the message has an address `a`

The basic principle behind MPI'a derived datatypes is:
– in a new MPI datatype, provide all of the information except the beginning address.

A general MPI datatype is a sequence of pairs

$$\{(t_0, d_0), (t_1, d_1), \cdots, (t_{n-1}, d_{n-1})\} \tag{1}$$

where t_k) is a basis MPI datatype and d_k is displacement in bytes.

For the above example:

`{(MPI_FLOAT,0),(MPI_FLOAT,16),(MPI_INT,24)}`

Type map: the sequence (1).

Extent: the span from the first byte to the last byte occupied at entries in the datatype, rounded up to satisfy alignment requirements.

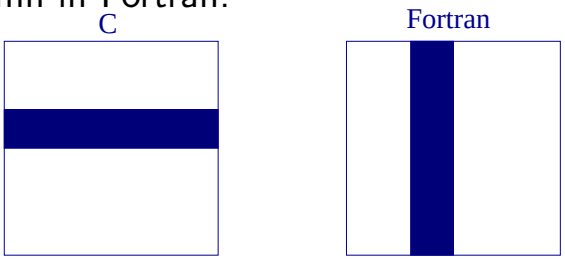
Example: Consider $Type = \{(double, 0), (char, 8)\}$.
Assume that doubles have to be aligned at addresses that are multiple of 8.

The extent of this type is 16.

The extent will be the same for $Type = \{(char, 0), (double, 1)\}$.

MPI provides a mechanism to build general user-defined data types.
However, the construction phase of these is expensive!
Thus, an application should use those many times to amortize the 'build' costs.

Example: We want to sent n number of contiguous elements, all of the same type. Consider an array `a(n,n)`. Say, we want to communicate a row in C and a column in Fortran.



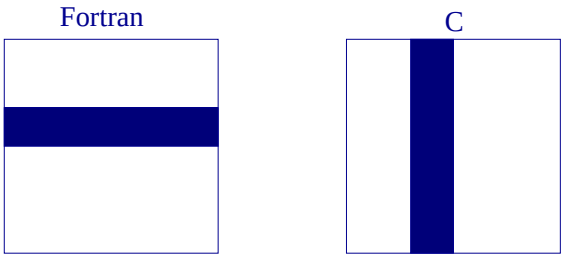
MPI_TYPE_CONTIGUOUS(count,oldtype,newtype)

oldtype 

count = 3

newtype 

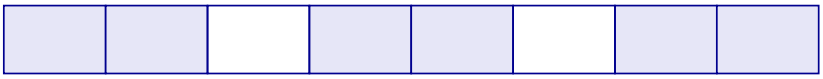
Example: Now, for the same array `a(n,n)`, we want to communicate a row in `Fortran` and a column in `C`. Both cases: not contiguous, but have a constant *stride*.



MPI_TYPE_VECTOR(count,blklen,stride,oldtype,newtype)

oldtype 

count = 3, blklen = 2, stride = 3

newtype 

`MPI_TYPE_VECTOR(n,1,n,MPI_DOUBLE,MPI_VDOUBLE)`

Example: Consider an already defined datatype `oldtype` which has type map $\{(double, 0), (char, 8)\}$.
A call to

```
MPI_Type_Vector(2,3,4,oldtype,newtype)
```

will create type map

{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40), ...
(double,64),(char,73),(double,80),(char,88),(double,96),(char,104)}

I.e., two blocks with three copies of the old type, with a stride 4×16 elements between the blocks.

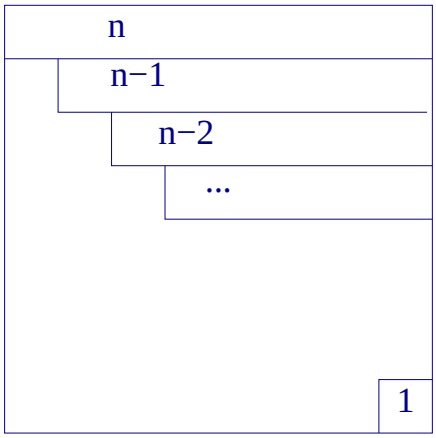
Example:
Consider again `oldtype` with type map $\{(double, 0), (char, 8)\}$.
A call to

```
MPI_Type_Vector(3,1,-2,oldtype,newtype)
```

will create type map

{(double,0),(char,8),(double,-32),(char,-24),(double,-64),(char,-56)}

Another scenario: Communicate the upper triangular part of a square matrix.



MPI_TYPE_INDEXED(count,array_of_blklen,array_of_displ,... oldtype,newtype)

oldtype 

count = 3, blklen = (2,3,1), displ=(0,3,8)

newtype 

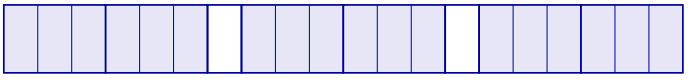
```
for (i=0; i<n; i++) {
     blk_len[i]=n-1;
     displ[i] = (n+1)*i;
}
MPI_Type_Indexed(n,blklen,displ,MPI:Float, &MPI_U);
MPI_Type_Commit(&MPI_U);
if (my_rank==0)
   MPI_Send(A,1,MPI_U,1,0,MPI_COMM_WORLD);
else  /* my_rank==1  */
   MPI_Recv(T,1,MPI_U,0,0,MPI_COMM_WORLD,&status);
end
```

OBS: Type-vector `MPI_Type_Vector(3,1,-2,...)` is inapplicable since the lengths of the portions are different.

MPI_TYPE_HVECTOR(count,blklen,stride,oldtype,newtype)

oldtype 

count = 3, blklen = 2, stride = 7

newtype 

Identical to MPI_TYPE_VECTOR, however the stride is given in bytes and not in elements.

'H' stands for *homogeneous*.

Allows for specifying overlapping entries.

Example: We have assumed that 'double' cannot start at displacements 0 and 4. Consider already defined datatype `oldtype` which has type map $\{(double, 0), (char, 8)\}$.
A call to

```
MPI_Type_Hvector(2,3,4,oldtype,newtype)
```

will create type map

{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40), ...
(double,4),(char,12),(double,20),(char,28),(double,36),(char,44)}

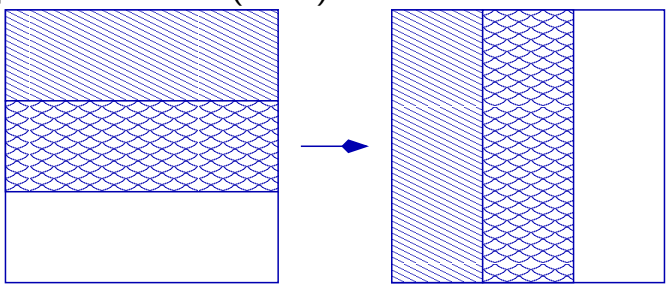
Example: Task: transpose a matrix:

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C transpose matrix a onto b

      CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)

C create datatype for one row
C (vector with 100 real entries and stride 100)
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
```

Example: Transpose a matrix (cont):

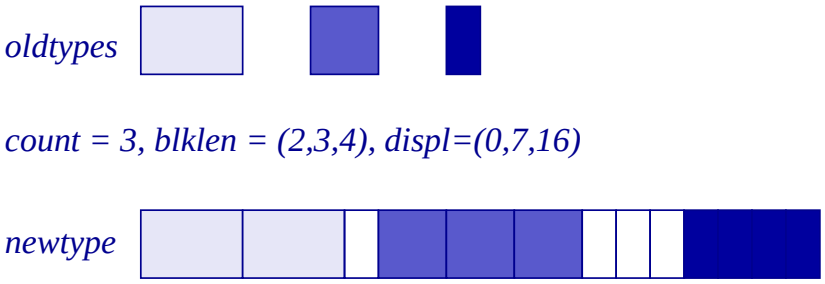
```
C create datatype for a matrix in row-major ordering
C (100 copies of the row datatype, strided 1 word apart;
C  the successive row datatypes are interleaved)

      CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, matrow, ierr)

      CALL MPI_TYPE_COMMIT(matrow, ierr)
C send matrix in row-major ordering and receive it in column-major order

      call MPI_SENDRECV(a,1,      matrow,  myrank,0,
                        b,100*100,MPI_REAL,myrank,0,MPI_COMM_WORLD,status,ierr)
```


Example: Transpose a matrix (cont):



NOTE!! The whole array is LOCAL for the process (processes).

MPI_TYPE_STRUCT(count,array_of_blklen,array_of_displ,array_of_types)



MPI_TYPE_STRUCT allows to describe a collection of data items of various elementary and derive types as a single data structure.

Data is viewed as a set of blocks, each of whih – with its own count and data type, and a location, given as a displacement.

OBS! The displacement need not be relative to the beginning of a particular structure. They can be given as absolute addresses as well.
In this case they are treated as relative to the starting address in memory, given as

MPI_BOTTOM

```
MPI_Bcast(MPI_BOTTOM,1,struct_type,0,comm)
```

Example: Another approach to the transpose problem (cont.)

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)

C transpose matrix a onto b

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
```


Example: Another approach to the transpose problem (cont.)

```
C create datatype for one row, with the extent of one real number
      disp(1) = 0
      disp(2) = sizeofreal
      type(1) = row
      type(2) = MPI_UB
      blocklen(1) = 1
      blocklen(2) = 1
      CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)

      CALL MPI_TYPE_COMMIT( row1, ierr)

C send 100 rows and receive in column major order
      CALL MPI_SENDRECV( a,100,    row1,    myrank, 0,
                         b,100*100,MPI_REAL,myrank, 0,
                         MPI_COMM_WORLD, status, ierr)
```

User-defined datatypes - struct

```
/* set up 4 blocks */
int          blockcounts[4]={50,1,4,2};
MPI_datatype types[4];
MPI_Aint     displs[4];
/* initialize types and displs with addresses of items */
MPI_Address(&cmd.display,    &displs[0] );
MPI_Address(&cmd.max,        &displs[1] );
MPI_Address(&cmd.min,        &displs[2] );
MPI_Address(&cmd.error,      &displs[3] );
types[0] = MPI_CHAR;
types[1] = MPI_INT;
types[2] = MPI_INT;
types[3] = MPI_double;

for (i=3; i>=0;i--)
     displs[i]-=displs[0];

MPI\_Type_struct(4,blockcounts,displs,types,&strtype);
MPI\_Type_comit(&strtype);
MPI\_Bcast(cmd,1,strtype,MPI\_COMM\_WORLD);
```


Datatype Constructors - additional

MPI_TYPE_COMMIT(data_type)
MPI_TYPE_FREE(data_type)
MPI_TYPE_EXTENT(data_type,extent)
Returns the extent of a datatype.
Can be used for both primitive and derived data types.

MPI_TYPE_SIZE(data_type,size)
Returns the total size (in bytes) of the entries in the type signature, associated with the data type, i.e., the total size of the data in the message that would be created with this datatype.

$\{(double, 0), (char, 8)\}$
Extent is equal to 16
Size is equal to 9.

```
          CALL MPI_Type_vector ( 1, 1, 1,
     >                           MPI_DOUBLE_PRECISION,
     >                           MPI_REAL_8, ierr )
          if ( ierr .NE. MPI_SUCCESS ) then
            stop
          end if
          CALL MPI_Type_commit ( MPI_REAL_8, ierr )
          if ( ierr .NE. MPI_SUCCESS ) then
            stop
          endif
c ------------ - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          CALL MPI_Type_vector ( sgridy*sgridz, 1, sgridx,
     >                           MPI_REAL_8,
     >                           type_fixed_x, ierr )
          CALL MPI_Type_commit ( type_fixed_x, ierr )

          CALL MPI_Type_vector ( sgridz, sgridx, sgridy,
     >                           MPI_REAL_8,
     >                           type_fixed_y, ierr )
          CALL MPI_Type_commit ( type_fixed_y, ierr )
```



```
c    --------- fetch from EAST: [xv(i,j,k) = x(i+distx,j,k)]
       if (NEWS27(1) .ne. 999) then
       call MPI_SENDRECV(xv(nanrx+1,1,1),1,type_fixed_x,NEWS27(1),1,
      >                  xv(nanrx,1,1),  1,type_fixed_x,NEWS27(1),2,
      >                   MPI_COMM_WORLD,status,ierr)
       endif
c    ---------- fetch from NORTH: [xv(i,j,k) = x(i,j+disty,k)]
       if (NEWS27(3) .ne. 999) then
       call MPI_SENDRECV(xv(1,nanry+1,1),1,type_fixed_y,NEWS27(3),3,
      >                  xv(1,nanry,1),  1,type_fixed_y,NEWS27(3),4,
      >                   MPI_COMM_WORLD,status,ierr)
       endif
```

```
if (my_rank==0)
   MPI_Send(mesage,send_count,send_type,1,0,comm);
else if (my_rank==1)
   MPI_Recv(mesage,resv_count,recv_type,0,0,comm);
```

Must send_type be identical to send_recv?

Type signature of the derived datatype: $\{t_0, t_1, \cdots, t_{n-1}\}$

Fundamental MPI rule:
the type signature of the sender and receiver must be compatible.

$\{t_0^s, t_1^s, \cdots, t_{n-1}^s\}$ $\{t_0^r, t_1^r, \cdots, t_{m-1}^r\}$

then $n \leq m$ and $t_i^s \equiv t_i^r$ for $i = 1, \cdots, n-1$.

Example: Array $A(10, 10)$, `float`
Task: Receive a column of it into a row of another array of the same size.

```
if (my_rank==0)
   MPI_Send(&A([0],[0]),1,col_type,1,0,comm,comm);
else if (my_rank==1)
   MPI_Recv(&A([0],[0]),10,MPI_Float,0,0,comm,&stat);
endif
```

Basic Linear Algebra Subroutines (BLAS)

The BLAS routines fall into three categories, depending on whether the operations involve vectors or matrices:

(i) vector or scalar operations - Level 1 BLAS;

(ii) matrix-vector operations - Level 2 BLAS;

(iii) matrix-matrix operations - Level 3 BLAS.

The BLAS 1 subroutines perform low granularity operations on vectors that involve one or two vectors as input and return either a vector or a scalar as output. In other words, $O(n)$ operations are applied on $O(n)$ data, where n is the vector length.
Some of the BLAS -1 operations:

$\mathbf{y} \longleftarrow a\mathbf{x} + \mathbf{y}$	vector update
$\mathbf{x} \longleftarrow a\mathbf{x}$	
$\mathbf{y} \longleftarrow \mathbf{x}$	vector copy
$dot \longleftarrow \mathbf{x}^T\mathbf{y}$	dot product
$nrm2 \longleftarrow \|\mathbf{x}\|_2$	vector norm

BLAS 2 perform operations of a higher granularity than BLAS Level 1 subprograms. These include matrix- vector operations, i.e., $O(n^2)$ operations, applied to $O(n^2)$ of data. The major operations:

$\mathbf{y} \longleftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$	
$\mathbf{y} \longleftarrow \alpha A^T\mathbf{x} + \beta\mathbf{y}$	
$\mathbf{y} \longleftarrow T\mathbf{x}$	T is a triangular matrix
$A \longleftarrow \alpha\mathbf{x}\mathbf{y}^T + A$	rank-one update
$H \longleftarrow \alpha\mathbf{x}\mathbf{y}^H + \bar{\alpha}\mathbf{y}\mathbf{x}^H + H$	rank-two update, H is hermitian
$\mathbf{y} \longleftarrow T\mathbf{x}$	multiplication by a triangular system
$\mathbf{y} \longleftarrow T^{-1}\mathbf{x}$	solution of a system with a triangular matrix

BLAS 3 are aimed at matrix-matrix operations, i.e., $O(n^3)$ operations, applied to $O(n^2)$ of data.
Some of the level-3 routines:

$C \longleftarrow \alpha AA^T + \beta C$	matrix rank-one update
$C \longleftarrow \alpha AB^T + \alpha BA^T + \beta C$	matrix rank-two update
$B \longleftarrow \alpha T^{-1} B$	solution of a system with a triangular matrix and many right-hand sides

BLACS aim at *ease of programming*, *ease of use* and *portability*.

BLACS serve a particular audience and operate on 2D rectangular matrices (scalars, vectors, square, triangular, trapezoidal matrices are particular cases).

Syntax: vXXYY2D
- v - the type of the objects (I,S,D,C,Z);
- XX - indicates the shape of the matrix (GE, TR)
- YY - the action (SD (send), RV, BS, BR (broadcast/receive))

```
vGESD2D(M, N, A, LDA, RDEST, CDEST)
vGERV2D(M, N, A, LDA, RDEST, CDEST)
vTRSD2D(UPLO, DIAG, M, N, A, LDA, RDEST, CDEST)
```

Here
$A(M, N)$ is the matrix
LDA - leading matrix dimension
RDEST - row index of destination process
CDEST - column index of destination process

Example of a broadcasting routine call:

```
vGEBS2D('scope', 'topology', M, N, A, LDA)
```

where

'scope'	can be	'column', 'row';
'topology'	can be	'hypercube', 'tree'

Linear Algebra problems

Operations on matrices and vectors.

- dense matrix linear algebra
- sparse matrix linear algebra

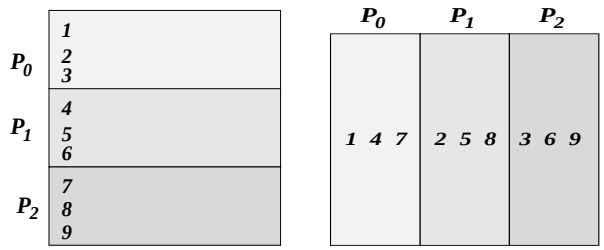
Dense matrix linear algebra

$A(n,n)$, $B(n,n)$, $C(n,n)$, $v(n,1)$, $w(n,1)$

Matrix-vector multiplication $\mathbf{w} = A\mathbf{v}$

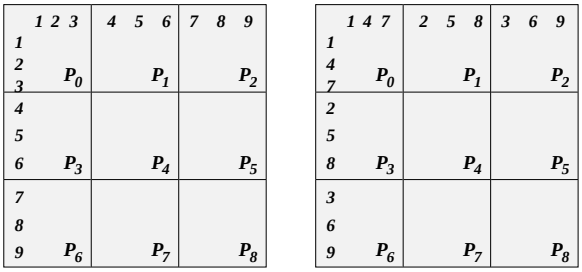
```
w = 0
do i=1,n
   do j = 1,n
      w(i) = w(i) + A(i,j)*v(j)
   enddo
enddo
-----------------------------------------------
do j=1,n
   do i = 1,n
      w(i) = w(i) + A(i,j)*v(j)
   enddo
enddo
```

Note: `w(:) = w(:) + v(j)*A(:,j)` is a vector operation!



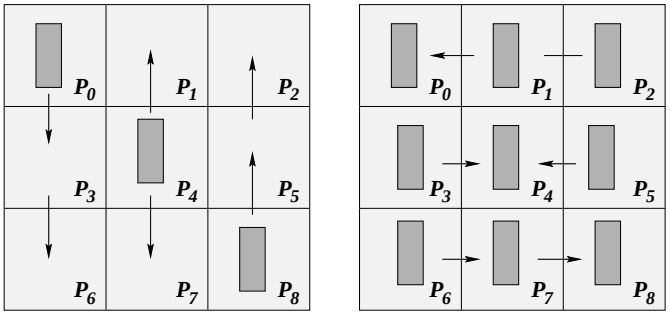
(a) block (b) cyclic

Striped partitionings



(c) block (d) cyclic

Checkerboard partitionings



(e) column-wise one-to-all broadcast (f) row-wise accumulation

Communications during matrix-vector multiplication for block checkerboard partitioning

$$\begin{bmatrix} A11 & A12 & A13 \\ A21 & A11 & A23 \\ A31 & A32 & A33 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} A11v_1 + A12v_2 + A13v_3 \\ A21v_1 + A22v_2 + A23v_3 \\ A31v_1 + A32v_2 + A33v_3 \end{bmatrix}$$

```
w = 0
do i=1,n
   do j = 1,n
      do k = 1,n
         C(i,j) = C(i,j) + A(i,k)*K(k,j)
      end
   enddo
enddo
```

Serial complexity: n^3

A11	*A12*	*A13*
A21	*A22*	*A23*
A31	*A32*	*A33*

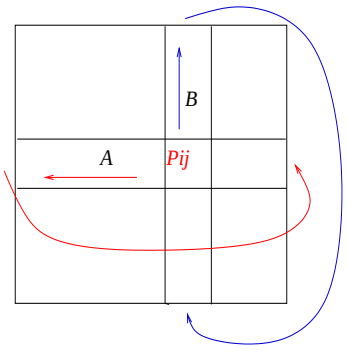
B11	*B12*	*B13*
B21	*B22*	*B23*
B31	*B32*	*B33*

Scenario 1:
All-to-all on each row of A
All-to-all on each column of B
Local multiplications and additions

Both communication and memory demanding!

Scenario 2: Cannon's algorithm
L.E. Cannon, *A cellular computer to implement the Kalman Filter Algorithm*, Ph.D. thesis, Montana State University, Bozman, MT, 1969.

Let A, B, C be $n \times n$ and the number of processors be p.
The matrices A, B and C are partitioned in blocks $(A^{ij}, B^{(ij)}, C^{(ij)})$.
whenever $A^{(ik)}$ and $B^{(kj)}$ happen to be in the processor (i, j), they are multiplied and accumulated into $C^{(ij)}$.



Cannon's algorithm

```
for i = 1 : n      % row-wise
    assign A^(ij) to processor P_{i,(i+j) mod n}
end
for j = 1 : n      % column-wise
    assign B^(ij) to processor P_{(i+j) mod n,j}
end
for k = 1 : n
    forall (i = 1 : n, j = 1 : n)
            C^(ij) = C^(ij) + A^(ik) * B^(kj)
            Left circular shift each block row of A
            Upward circular shift each block column of B
    end
end
```


Assume that each processor holds blocks of size $(\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}})$. The algorithm requires $4p-2$ communication steps, during each $\dfrac{n^2}{p}$ amount of words is transferred. Thus, for the parallel time we obtain

$$T_p = \frac{n^3}{p} + (4p-2)\frac{n^2}{p},$$

compared to n^3 in the serial case. For the speedup and efficiency the figures are (replacing $4p-2$ by $4p$)

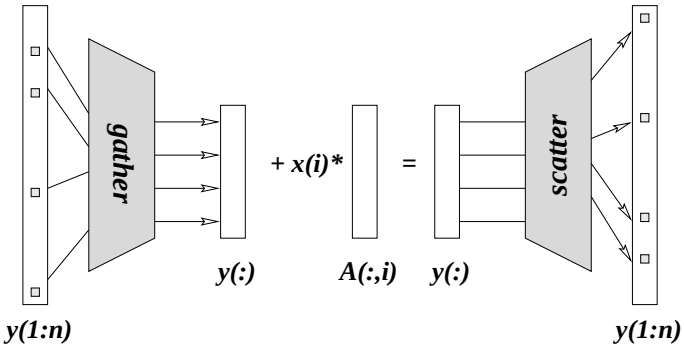
$$S = \frac{1}{\dfrac{1}{p} + \dfrac{4}{n}}, \quad E = \frac{1}{1 + \dfrac{4p}{n}}. \tag{2}$$

Relation (2) shows that if p is such that $n \geq 36p$, for instance, the efficiency becomes above 0.9.

However, this algorithm has the disadvantage that it does not take into account whether the data layout it requires is suitable for other matrix operations.

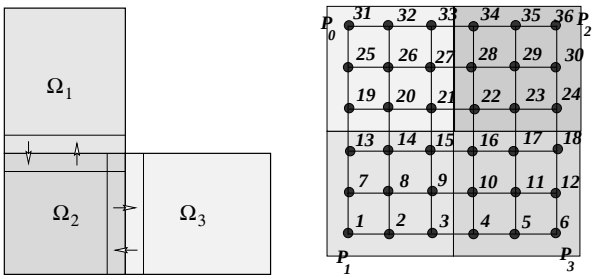
Sparse matrices

Shared memory



$$Y(:) = Y(:) + x(i) * A(i,:)$$

Distributed memory

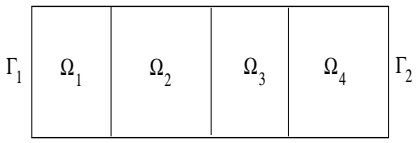


(g) (h)

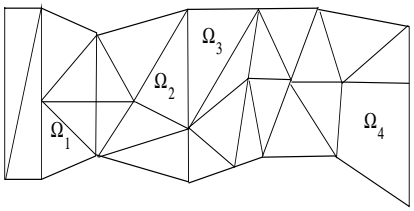
Grid-wise mapping of a discrete problem onto distributed memory computer

(Block-)Tridiagonal Systems

$$A = \begin{bmatrix} A_{11} & A_{12} & & 0 \\ A_{21} & A_{22} & A_{23} & \\ & \ddots & \ddots & \ddots \\ 0 & & A_{n,n-1} & A_{n,n} \end{bmatrix} \text{ or } A = \text{tridiag}\,(A_{i,i-1}, A_{i,i}, A_{i,i+1}).$$



(i) Subdomain division and ordering.



(j) Subdomain division of a network.

Given a tridiagonal matrix A. The usual way is to factorize it as $A = LU$ and then solve $A\mathbf{x} = \mathbf{b}$ as
$L\mathbf{z} = \mathbf{b}$ and $U\mathbf{x} = \mathbf{z}$.

Both factorization (Gauss elimination) and the solution of triangular (in this case, lower- and upper-bidiagonal) systems is PURELY SEQUENTIAL by its nature !!!

forward substitution: $L\mathbf{z} = \mathbf{b}$, i.e.,

$$z_1 = b_1$$
$$z_i = b_i - \sum_{k=1}^{i-1} l_{i,k} z_k, \; i = 2, 3, \cdots, n.$$

backward substitution: $U\mathbf{x} = \mathbf{z}$, i.e.,

$$x_n = z_n$$
$$x_i = z_i - \sum_{k=i+1}^{n} u_{i,k} x_k, \; i = n-1, \cdots, 1.$$

$$\begin{bmatrix} * & 0 & 0 & 0 \\ * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

In order to gain some **parallelism when solving linear recursions**, some special techniques have been studied:

- **Multifrontal solution methods**
- **Odd-even elimination methods**
- **Recursive doubling**
- **Divide-and conquer methods**

The beginning of the end for Day 3:

The Conjugate Gradient method

Assume A and $\mathbf{b}$ are distributed and an initial guess $\mathbf{x}^{(0)}$ is given, which is replicated.

$$
\begin{array}{rrcl}
 & \mathbf{g}^{(0)} & = & \mathbf{b} - A\mathbf{x}^{(0)} \\
 & \mathbf{r} & = & replicate(\mathbf{g}^{(0)}) \\
 & \mathbf{d}^{(0)} & = & -\mathbf{r} \\
 & \delta_0 & = & (\mathbf{g}^{(0)}, \mathbf{r}^{(0)}) \\
\textit{For } k = 0, 1, \cdots & \textit{until convergence} & & \\
(1) & \mathbf{h} & = & A\mathbf{d}^{(k)} \\
(2) & \tau & = & \delta_0/(\mathbf{h}, \mathbf{d}^{(k)}) \\
(3) & \mathbf{x}^{(k+1)} & = & \mathbf{x}^{(k)} + \tau\mathbf{d}^{(k)} \\
(4) & \mathbf{g}^{(k+1)} & = & \mathbf{g}^{(k)} + \tau\mathbf{h} \\
(5) & \mathbf{r} & = & replicate(\mathbf{g}^{(k+1)}) \\
(6) & \delta_1 & = & (\mathbf{g}^{(k+1)}, \mathbf{r}) \\
(7) & \beta & = & \delta_1/\delta_0, \delta_0 = \delta_1 \\
(8) & \mathbf{d}^{(k+1)} & = & \mathbf{r} + \beta\mathbf{d}^{(k)}
\end{array}
$$