Who Needs Communication

Communications between tasks depends upon problem:

- No need for communication:
  - Some problems can be decomposed and executed in parallel with virtually no need for tasks to share data.
  - Often called embarrassingly parallel as they are straight-forward. Very little inter-task communication is required.
- Need for communication:
  - Most parallel applications not so simple $\rightarrow$ require tasks to share data with each other.
  - Changes to neighboring data has a direct effect on that task's data.

1.1.4. Further Keywords

- Load balancing: Job should be distributed such that all processors are busy all the time!
  avoid idle processors ⇒ efficiency
- Dead lock: Two or more processors waiting indefinitely for event that can be caused only by one of themselves.

  → Each processor waiting for results of the other processor!

- Race condition: Non-deterministic outcome/results depending on chronology of parallel tasks.

For Synchronization

- Barrier:
  - Implies that all tasks are involved.
  - Each task performs work until reaching barrier. It then stops.
  - When last task reaches barrier, all tasks are synchronized.
- Lock/semaphore:
  - Can involve any number of tasks. Can be blocking or non-blocking.
  - Used to serialize/protect access to global data or code section.
  - Only one task at a time may use (own) the lock/semaphore/flag.
  - The first task to acquire the lock 'sets' it.
  - Task can then safely access the protected data or code.
  - Other tasks can attempt to get lock but must wait until it is released.
- (Synchronous communication operations:)
  - Involves only tasks executing a communication operation
  - When task performs a communication operation, coordination with participating tasks is required.
  - Discussed previously in the communications section.

Granularity: Computation/Communication Ratio

- Qualitative measure of ratio of computation to communication.
- Periods of computation separated from periods of communication by synchronization events.

Granularity of Parallelism

- Fine-grain Parallelism:
  - \+ Small amounts of computational work done between communication events.
  - \- Low computation to communication ratio.
  - \+ Faciliates load balancing.
  - \- Implies high communication overhead, less opportunity for performance enhancement.
  - \- If granularity too fine $\rightarrow$ overhead for communication and synchronization between tasks larger than for computation.
- Coarse-grain Parallelism:
  - \+ Large amounts of computational work done between communication/synchronization events.
  - \+ High computation to communication ratio.
  - \+ Implies more opportunity for performance increase.
  - \- Harder to achieve efficient load balancing.

Scalability

- How well does a solution to a given problem work when the size of the problem increases.
- How well does a parallel solution to fixed-size problem work when the number of processors increases.
- Strong scaling (Amdahls's law): **Fixed total problem size.** How does the solution time vary with the number of used processors.
- Weak scaling (Gustafson's law): **Fixed problem size per processor.** How does the solution time vary with the number of used processors.

Parallel Overhead

Amount of time required to coordinate parallel tasks, as opposed to doing useful work.

Parallel overhead can include factors such as:

- Task start-up time and termination time
- Synchronizations
- Data communications
- Software overhead by parallel compilers, libraries, tools, OS, ...

Further Keywords

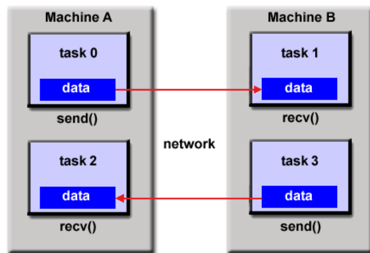
- Massively Parallel: HW that comprises many PEs—meaning of ”many” keeps increasing.
  Largest parallel computer currently $\sim 1.6 \cdot 10^6$ cores.
- Cluster Computing: Use of a combination of commodity units (processors, networks or SMPs) to build a parallel system.
- Supercomputing/High Performance Computing: Use of the world’s fastest, largest machines to solve large problems.
- Grid Computing: is the combination of computer resources from multiple administrative domains applied to a common task.
- Cloud Computing: is the provision of dynamically scalable and often virtualised resources (data, software/algorithms, computing power, ...) as a service over the Internet on a utility basis.

1.1.5. Message Passing for Distributed Memory: MPI

Message Passing Interface MPI:
Communication subroutine library for C,C++, Fortran

Compiling: `mpicc <options> prog.c`

Start: `mpirun -arch<architecture> -np<np> prog`



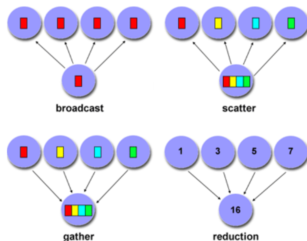
Synchronous vs. Asynchronous Communication

- Synchronous communication:
  - Requires some type of ”handshaking” between tasks that are sharing data. (explicitly structured in code by programmer)
  - Often referred to as blocking communications since other work must wait until the communications have completed.
- Asynchronous communication:
  - Allow tasks to transfer data independently from one another.
  - Are often referred to as non-blocking communications.
  - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Important MPI Commands

Commands:

- MPI_Send – MPI_Recv
- MPI_Barrier
- MPI_Bcast – MPI_Gather
- MPI_Scatter – MPI_Reduce



Introduction to MPI with commands and application to numerical examples→ Tutorial.

(Some) Important MPI Variables

- `MPI_COMM_WORLD`: Description of the environment
- `myrank`: number of actual process
- `MPI_CHAR, MPI_DOUBLE`: type of variables
- `MPI_Status, MPI_Request`: information on communication progress identifiable by request

Blocking vs. Nonblocking

- `MPI_Send`: Blocking in the sense that the sending process can only proceed if the receiving has started.
- `MPI_Ssend`: Synchronous send. Will only complete if matching receive is posted and has started to receive the message.
- `MPI_Rsend`: Ready send. For large data sets. Starts with sending only in case receiver is prepared.
- `MPI_Isend`: Nonblocking. Careful with overwriting data that should be sent!
- `MPI_Bsend`: Nonblocking buffered send. If no matching receive posted, MPI buffers outgoing message.

Collective Communication

- Synchronization of tasks: Barrier
- Broadcast
- Gather
- Scatter
- Reduction

- All tasks are calling the same function (e.g. broadcast)
- The exact data size has to be given
- There can be only one collective operation active
- All collective functions are blocking
- Synchronization only by barrier (`MPI_Barrier`)

1.1.6. Shared Memory Communication

Avoid data access from different PEs on same data at same time!

Synchronisation by

- barrier: each parallel thread waits until all are finished
- critical sections with reduced access

Tool: OpenMP with additional commands to programming languages C, C++, or FORTRAN. OpenMP compiler!

OpenMP is a set of compiler directives.

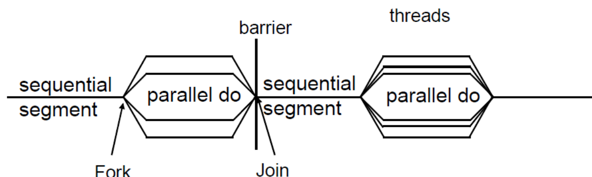
Pragma:

```
#pragma omp parallel          c$omp parallel
{...                          ...
}                             c$omp end parallel
in C,C++                      in FORTRAN
```

Also: `c$omp critical` and `c$omp end critical`
`c$omp private` `c$omp barrier`

OpenMP

Parallel regions



General definition of a parallel region:

```
c$omp parallel
...
c$omp end parallel
```

Threads

- OpenMP program generates POSIX Threads as elementary basic processes. Master thread (sequential) and parallel independent threads.
- Thread as independent process:
  - initiated by operating system
  - concurrent (on same CPU) with all other processes/threads
  - may pause, switch to other processor,
- Problems:
  - Data access (global memory, ev. generate private copies)
  - Time schedule (no information on thread start/end time)
- Thread safety:
  - Subroutines have to be independent and safe units.
  - Call of a subroutine by independent threads should not introduce unwanted side effects! (Race conditions, dead locks,...)
- No communication necessary! Control of common memory access.

Example for Race Condition

```
int called=0, *dat;
void not_thread_safe(n) {
    if(called==0) {
        called=1;
        dat=(int *)malloc(sizeof(int)*n);
    }
}
```

If different threads read variable `called` at the same time
$\rightarrow$ for all these threads the if-condition is satisfied and all they will allocate new memory for pointer `dat`!

Synchronisation via Semaphores

- A semaphore is a signal to coordinate processes, based on two operations:
  - `lock (P)`
  - `unlock (V)`
- Semaphore variable can have two possible values: locked or unlocked
- When semaphore is unlocked by the authorized process, other processes are free to access and lock it.
- If process tries to lock an unlocked semaphore then semaphore gets locked until the process unlocks it.
- If process tries to lock an already locked semaphore it has to wait until the previous process unlocks the semaphore.
- In OpenMP: semaphore indirectly usable;
  similar+direct: `critical` section

Example for Semaphores

```
Semaphore s
parallel do i=1,n
    s.P             // lock
    x=x+1
    s.V             // unlock
end parallel do
```

Without lock, one process could try to read x while another one is incrementing x. This would lead to a classical race condition and a nondeterministic behaviour of the code.

The steps in the loop are:

```
Read x
Increment x
Write x
Increment loop
```

Which value will be read by the process depends on step in loop.

Example for Dead Lock

```
call omp_init_lock(var)
c$omp parallel sections
c$omp section
    call omp_set_lock(var)
    ... (work)
    if (ival > tol) then
        call omp_unset_lock(var)
    else
        call error(ival)
    end if
c$omp section
    call omp_set_lock(var) // var potentially never unlocked!
    ... (work2 depends on work)
    call omp_unset_lock(var)
c$omp end parallel sections
```

Race condition can lead to several existing variables locked
$\rightarrow$ another thread waits infinitely long for unlocked variable.

Critical Sections in OpenMP

A critical section is a section of code in which only one subcomputation is active

In our previous example only one statement

```
x = x+1
```

can be active between `lock(s)` and `unlock(s)`.

Therefore only one processor will be in a critical section at one time!

Critical Sections in OpenMP

Explicit synchronization in OpenMP via the

```
c$omp critical
```

directive, which acts like a guard (or mutex = mutual exclusion) at the beginning of a critical section (lock/unlock mutex).

The end is marked by `c$omp end critical`

```
c$omp parallel do
do i = 1, n
  c$omp critical
  x = x + 1
  c$omp end critical
end do
```

Barrier in OpenMP

- A barrier is a point in a parallel code where all processors stop in order to synchronize.
- All processors wait there until all others have arrived.
- Once they have all arrived, they are all released to continue in the code.

Typical example:

```
parallel do i=1,n
    "local part of comput., each processor independent in parallel"
    call barrier()
    "all using results of other processors from previous part"
end parallel do
```

- After all reached barrier one can be sure that all other processors have computed their partial results and it is safe to use these results.

Loop Tiling in OpenMP

- OpenMP directives provide instructions to the compiler, but the directives are not translated into code.
- Simple case: parallel do instruction implies that the loop following it is to be subdivided into as many separate threads as are avaliable.

```
c$omp parallel do
    do i = 2, n-1
        newx(i) = .5*( x(i-1) + x(i+1) - dx2*x(i) )
    enddo
c$omp end parallel do //Synchronizing barrier, can be omitted
c$omp parallel do
    do i = 2, n-1
        x(i) = newx(i)
    enddo
c$omp end parallel do //can be omitted
```

Private Variables in OpenMP

- `c$omp parallel do private (a,b,i,j,x,y)`: parallel statement includes a list of private variables that are local in loop.
- This is similar to adding an extra array index to the variable to make it different for each iteration of the loop.
- Private variables can be safely used if there is no write statement involved with the private variables.

```
do i = 1, 100
     if ( parity(i) .EQ. 1 )
          temp = i  // temp can not be privatized safely
     endif
     if ( parity(i) .EQ. 0 )
          temp = 0
     endif
enddo
```

Examples for Use of Private Variables

```
c$omp parallel do private(TEMP,I)
do I = 1, 100
    TEMP = I
    A(I) = 1.0 / TEMP
enddo
c$omp end parallel do

c$omp parallel do private(k)
do k = 0, P-1
    localsum(k) = 0.0
    do i = 1 + k*(N/P),  (k+1)*(N/P)
        localsum(k) = localsum(k) + x(i)
    enddo
enddo
c$omp end parallel do
```

Decomposition Choices in OpenMP

- `c$omp parallel do schedule(static)`
  indicates a static block decomposition of the iteration space
  blocksize = iteration space / number of processors
- `c$omp parallel do schedule(static, chunksiz)`
  indicates a static block decomposition of the iteration space
  where the blocksize is given by `chunksiz`
- `c$omp parallel do schedule(dynamic, chunksiz)`
  indicates that chunks of loop indices should be allocated to
  processors on a first-available basis
- `c$omp parallel do schedule(guided, chunksiz)`
  starts with a large size of chunks each processor takes, and then
  exponentially reduces the chunk size down to `chunksiz`.

Reductions in OpenMP

```
#pragma omp parallel for reduction(+:x,y)
for (i = 0; i < n; i++) {
    x += b[i];
    y = sum(y,c[i]);
}
```

Syntax: `reduction(operator:list)`

Compare `MPI_Allreduce`

How Many Processes?

```
call omp_set_num_threads(inthreads)
```

Designates the number of processes via inthreads.

The number of threads can be determined with

```
nThreads = omp_get_num_threads( )
```

OpenMP Compilers

- GNU gcc, compile with `-fopenmp`
- IBM XL C/C++/Fortran, Sun C/C++/Fortran, compile with `-xopenmp`
- Intel C/C++/Fortran, compile with `-Qopenmp` (Windows) or `-openmp` on Linux

Test code:

```
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    printf("Hello from thread %d, nthreads %d\n",
            omp_get_thread_num(), omp_get_num_threads());
}
```

1.1.7. Important Factors of Communication

- Cost of communication:
  - Inter-task communication implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communication frequently requires some type of synchronization between tasks, which can result in tasks spending time ”waiting” instead of doing work.
  - Competing communication traffic can saturate available network bandwidth, further aggravating performance problems.

Latency vs. Bandwidth

- Latency:
  - Time to send a minimal (0 byte) message from point A to point B.
  - Commonly expressed in microseconds.
- Bandwidth:
  - Amount of data that can be communicated per unit of time.
  - Commonly expressed in megabytes/sec or gigabytes/sec.
  - Sending many small messages can cause latency to dominate communication overheads.
  - Often more efficient to package small messages into larger message, thus increasing the effective communications bandwidth.

Visibility of Communication

- With Message Passing Model, communication is
  - explicit
  - generally visible
  - and under the control of the programmer.
- With shared memory model, communication is
  - not transparent to programmer.
  - Programmer may not even be able to know how and when inter-task communications are being accomplished.

1.2. Numerical Problems

- Operations for vectors $x, y \in \mathbb{R}^n$
- Dot or inner product

$$x^T y = (x_1, \ldots, x_n) \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \sum_{i=1}^{n} x_i y_i$$

- Sum of vectors

$$x + \alpha y = \begin{pmatrix} x_1 + \alpha y_1 \\ \vdots \\ x_n + \alpha y_n \end{pmatrix}$$

- Outer product

$$xy^T = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} (y_1, \ldots, y_m) = \begin{pmatrix} x_1 y_1 & \cdots & x_1 y_m \\ \vdots & \ddots & \vdots \\ x_n y_1 & \cdots & x_n y_m \end{pmatrix}$$

Matrix Product

$$\underbrace{\begin{pmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{pmatrix}}_{A \in \mathbb{R}^{n \times k}} \underbrace{\begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{k1} & \cdots & b_{km} \end{pmatrix}}_{B \in \mathbb{R}^{k \times m}} = \underbrace{\begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nm} \end{pmatrix}}_{C=A\cdot B \in \mathbb{R}^{n \times m}}$$

$$c_{ij} = \sum_{r=1}^{k} a_{ir} b_{rj}, \qquad i = 1, \ldots, n; \qquad j = 1, \ldots, m$$

$$c_{ij} = A(i,:)B(:,j) = \sum_{r=1}^{k} A(i,k)B(k,j)$$

Tensors: a_{ijk}, b_{rstj}. Product via summation of common index i.

Alternative Matrix Products

- Hadamard product:

$$A \in \mathbb{R}^{n\times k}, \quad B \in \mathbb{R}^{n\times k}, \quad C = A \circ B \in \mathbb{R}^{n\times k}$$

$$c_{ij} = a_{ij} \cdot b_{ij}, \qquad \text{for } i = 1, \ldots, n;\ j = 1, \ldots, k$$

- Kronecker (Tensor) product:

$$A \in \mathbb{R}^{n\times m}, \quad B \in \mathbb{R}^{s\times t}, \quad C = A \otimes B \in \mathbb{R}^{ns\times kt}$$

$$C = \begin{pmatrix} a_{11}B & \cdots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \cdots & a_{nm}B \end{pmatrix}$$

$$(a_1 \ \ a_2) \otimes (b_1 \ \ b_2 \ \ b_3) = (a_1b_1 \ \ a_1b_2 \ \ a_1b_3 \mid a_2b_1 \ \ a_2b_2 \ \ a_2b_3)$$

Solving Linear Equations

- Triangular system:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ddots &= \vdots \\ a_{nn}x_n &= b_n \end{aligned}$$

- Solution:

$$x_n = \frac{b_n}{a_{nn}} \Rightarrow a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = b_{n-1}$$

$$\Rightarrow x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n-1}}$$

- Algorithm: for $j = n, \ldots, 1$: $\quad x_j = \frac{b_j - \sum_{k=j+1}^{n} a_{jk}x_k}{a_{jj}}$

Eigenvalues

- Eigenvalue λ with eigenvector $u \neq 0$: $Au = \lambda u$
- For an $n \times n$-symmetric (Hermitian) matrix A there exist n pairwise orthogonal eigenvectors u_j with real eigenvalues $\lambda_j,\ j = 1, \ldots, n$.

$$Au_j = \lambda_j u_j, \quad j = 1, \ldots, n$$

$$U := (u_1, \ldots, u_n), \qquad \Lambda := \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$$

$$AU = U\Lambda \ \Rightarrow \ U^H AU = \Lambda$$

- Eigenvector basis describes optimal coordinate system!
- SVD and norms.

1.3. Data Dependency Graphs

General problem: Data dependency

Compute:

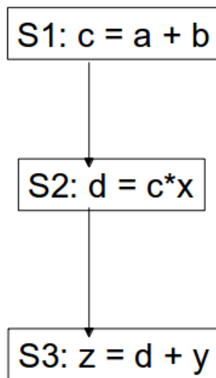
1. $c = a + b$
2. $z = c \cdot x + y$

Obviously, 2. can be computed only after 1.

Example: Self–referential loop:

```
for(i=1;i<=n;i++)
    a[i] = b[i] * a[i-1] + c[i]
```

Graphical Representation of Data Dependency

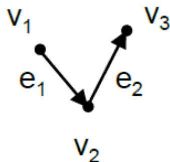


This will be further analysed in the following chapter.

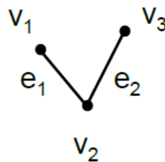
1.3.1. Graphs and Computations

Graph: $G = (V, E)$ with vertices v_i in V and edges $e_{ik} = (v_i, v_k)$ in E.

Directed graph: $e_{ik} \neq e_{ki}$



Undirected graph: $e_{ik} = e_{ki}$



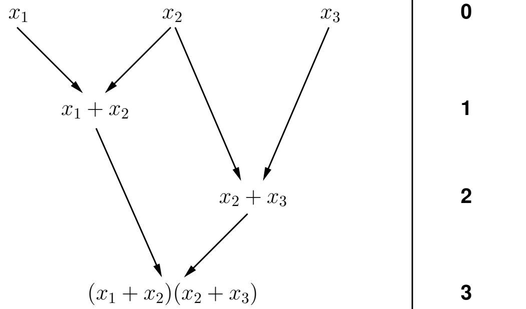
Example: Tree, star, ...

Example: Nonlinear Evaluation

Computation of $(x_1 + x_2)(x_2 + x_3) = x_1 x_2 + x_2^2 + x_1 x_3 + x_2 x_3$

Representation of the computational flow by a graph:

Sequential computation

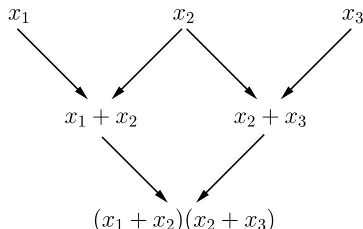


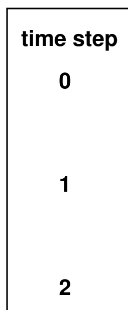
Sequential computation takes 3 time steps

Example: Nonlinear Evaluation

Computation of $(x_1 + x_2)(x_2 + x_3) = x_1 x_2 + x_2^2 + x_1 x_3 + x_2 x_3$

Parallel computation





Parallel computation takes 2 time steps

Example: Nonlinear Evaluation

Computation of $x_1 x_2 + x_2^2 + x_1 x_3 + x_2 x_3$

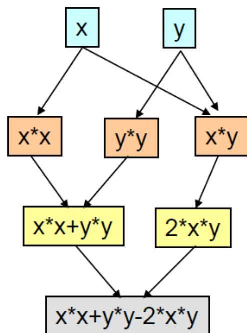
Parallel computation

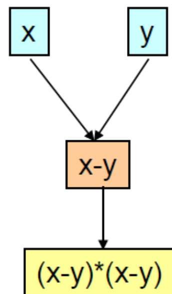
time step	
0	x_1, x_2, x_3
1	x_1x_2, x_2^2, x_2x_3, x_1x_3
2	$x_1x_2 + x_2^2$, $x_2x_3 + x_1x_3$
3	$x_1x_2 + x_2^2 + x_2x_3 + x_1x_3$

Parallel computation takes 3 time steps!

Further Example

Computation of $f(x, y) = x^2 - 2xy + y^2 = (x - y)^2$





3 time steps in parallel

2 time steps sequentially!