Introduction to Parallel Computing

Dr. Joachim Mai

Apr 2012

Content

Introduction

- Motivation: Why Parallel Programming
- Memory architectures (shared memory, distributed memory)
- Available Hardware
- Programming Models
- Designing Parallel Programs
- Costs of Parallel Programs

OpenMP

Intro into OpenMP with examples and exercises

MPI

Intro into MPI with examples and exercises

Why use Parallel Computing

The Universe is parallel

Parallel computing is just the next step of serial computing to describe systems which are intrinsically parallel.

INTERSECT

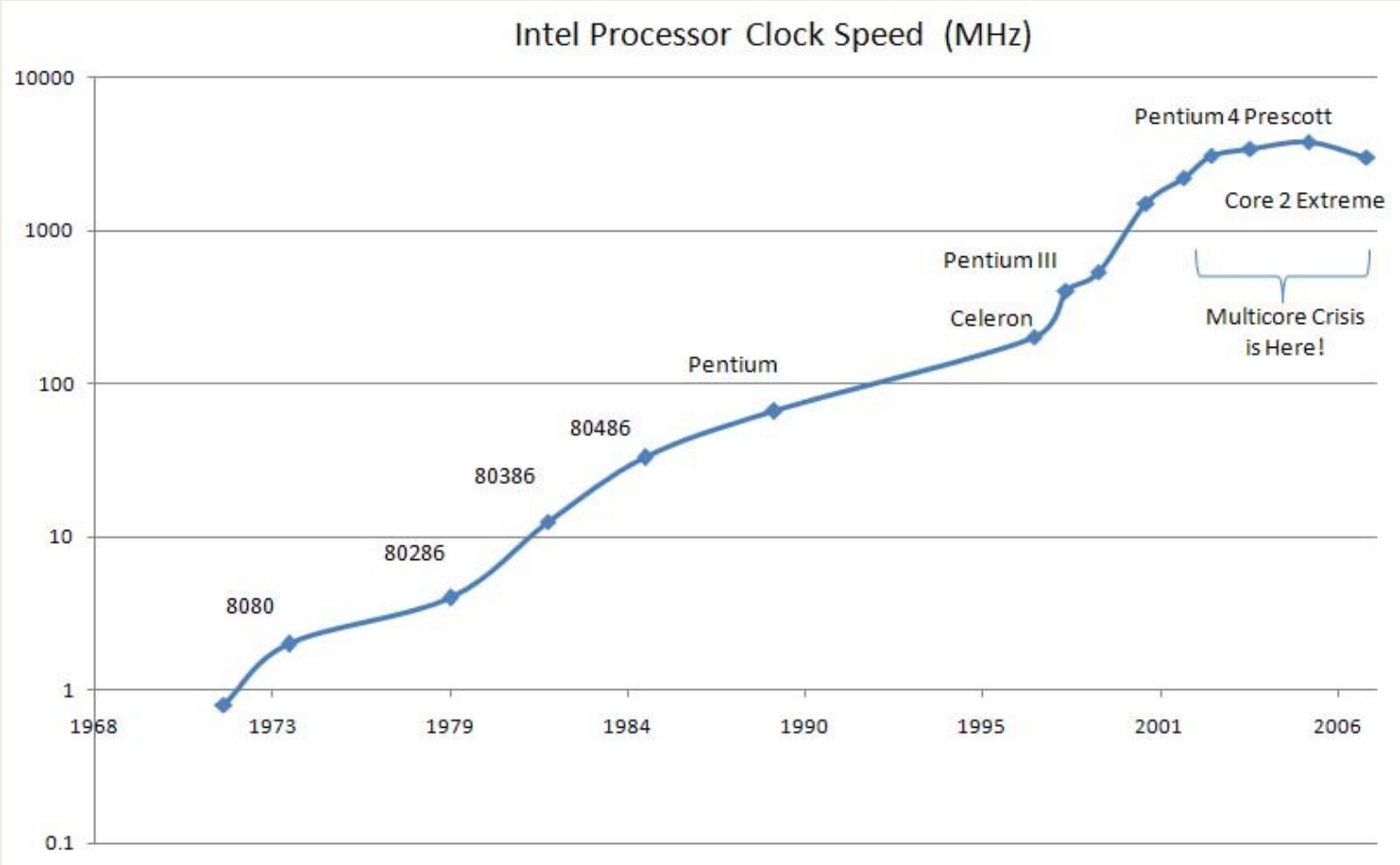
Parallel Computing

From Top500 (June 2010):

Name	Institute	No of cores
Jaguar	Oak Ridge	224,162
Nebulae	China	120,640
Roadrunner	DOE	122,400
Kraken	Comp. Sci.	98,928

Massive parallel machines

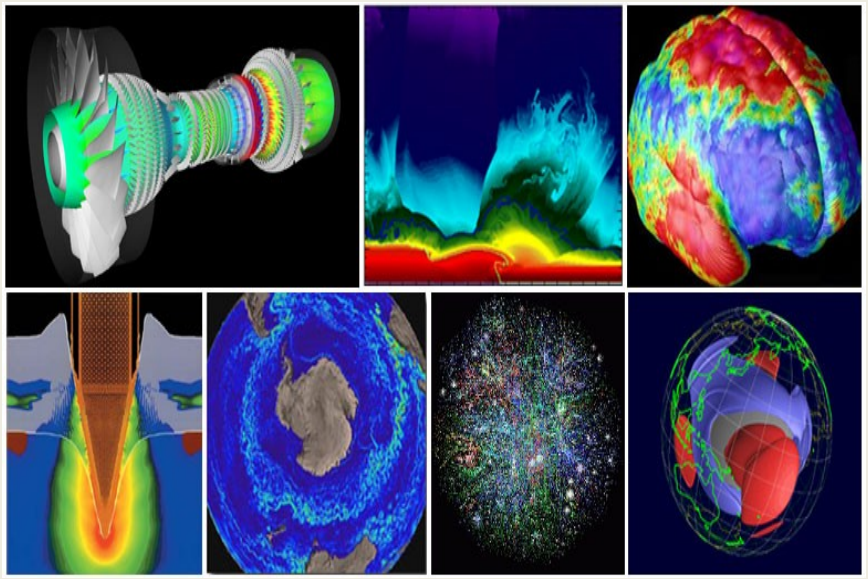
Clock Speed



Almost no frequency increase since 2000!

INTERSECT

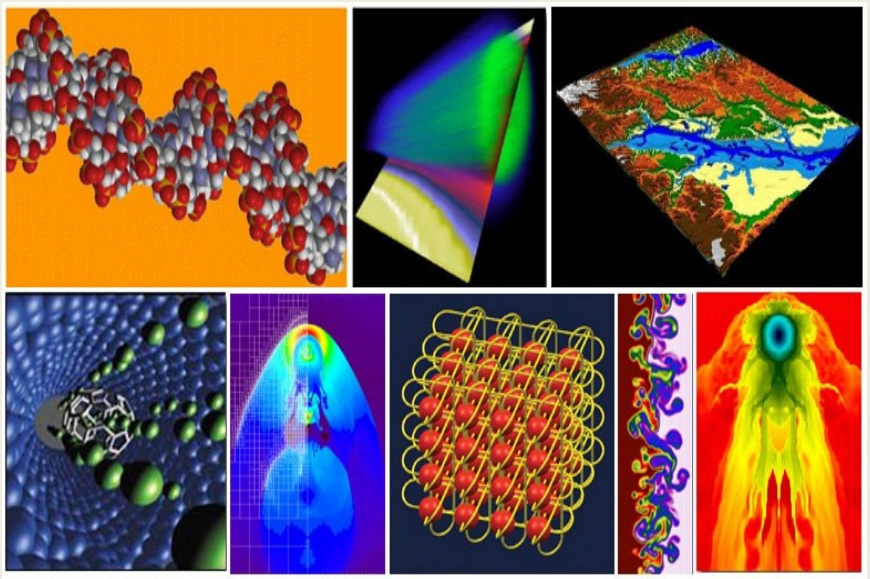
Uses for Parallel Computing



Scientific uses:

- Quantum Chemistry
- Solid State Physics
- Earth Sciences
- Mechanical Engineering
- Many more

Uses for Parallel Computing



Commercial uses:

- Data mining
- Financial modeling
- Pharmaceutical design
- Oil exploration
- Many more

What can Parallel Computing do?

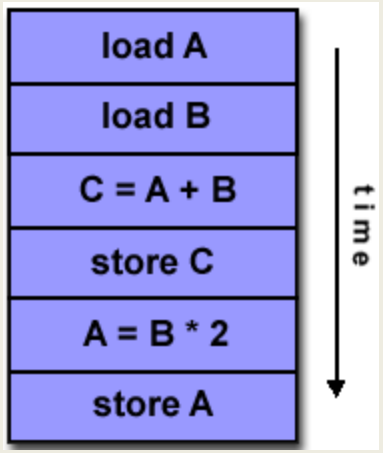
- Solve larger problems (Grand Challenges)
- Use non-local resources (Seti@Home)
- Solve problems quicker (Weather forecast)
- Save money (Stock transactions)
- Etc.

Flynn's Classical Taxonomy

1) SISD: Single Instruction, Single Data

A serial (non parallel computer)

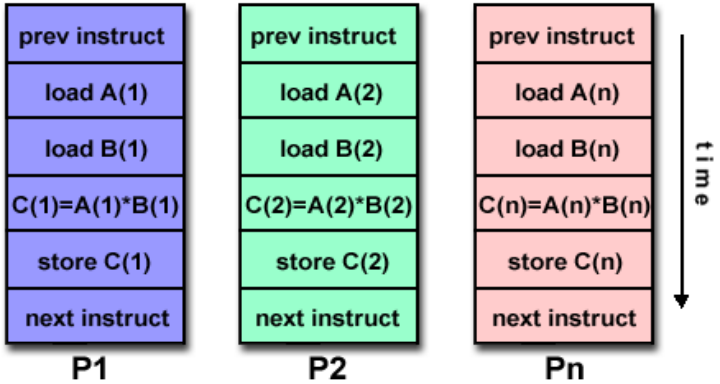
Only one instruction is used on a single data stream.

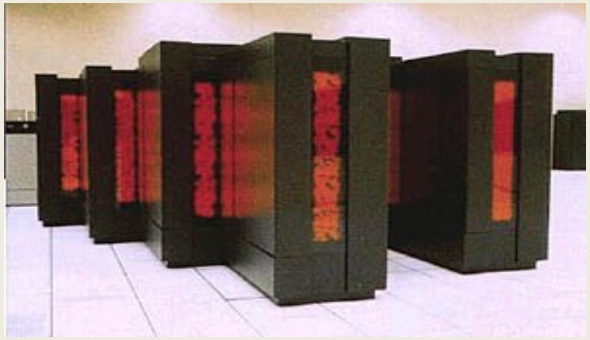


Flynn's Classical Taxonomy

2) SIMD: Single Instruction, Multiple Data

One instruction is used on several data.



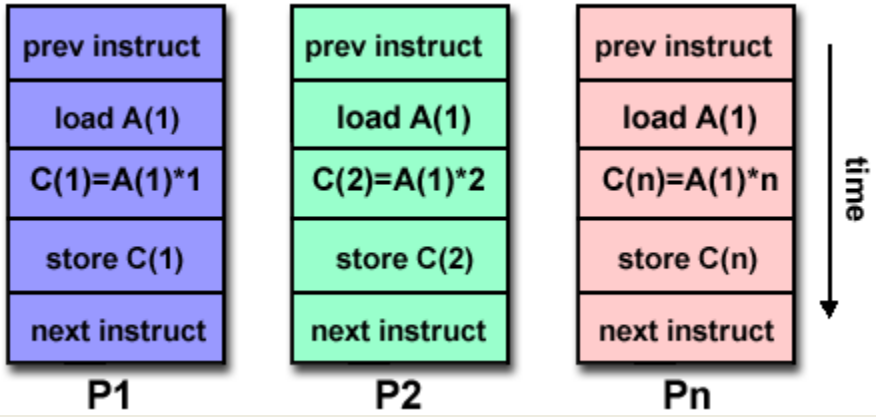


Flynn's Classical Taxonomy

3) MISD: Multiple Instructions, Single Data

Several instructions are used on a single data stream.

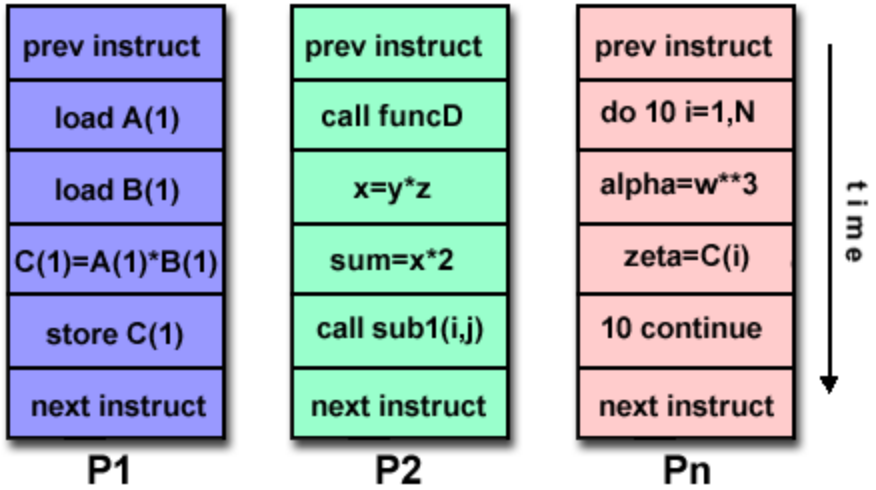
Only few computer ever existed.



Flynn's Classical Taxonomy

4) MIMD: Multiple Instructions, Multiple Data

Every processor might use different instructions on different data sets.

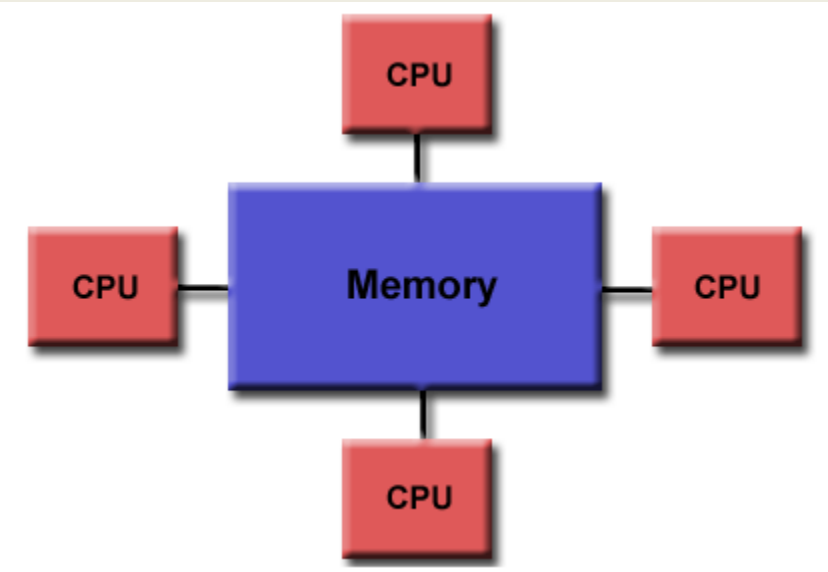


Memory Architectures

Shared memory architecture:

Uniform Memory Access (UMA)

Sometimes ccUMA (cache coherent)

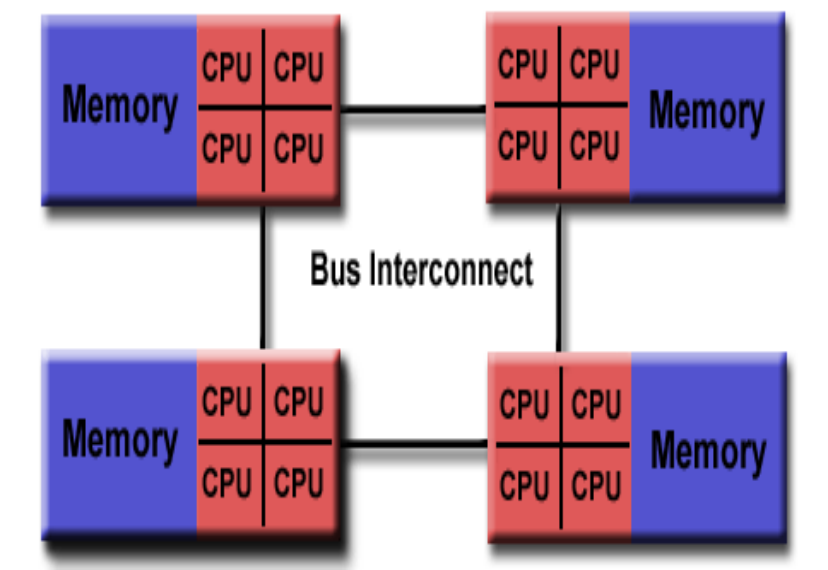


Memory Architectures

Shared memory architecture:

Non-Uniform Memory Access (NUMA)

Sometimes ccNUMA (cache coherent)



INTERSECT

Memory Architectures

Advantages of Shared Memory:

- Global address space (user friendly)
- Fast data sharing

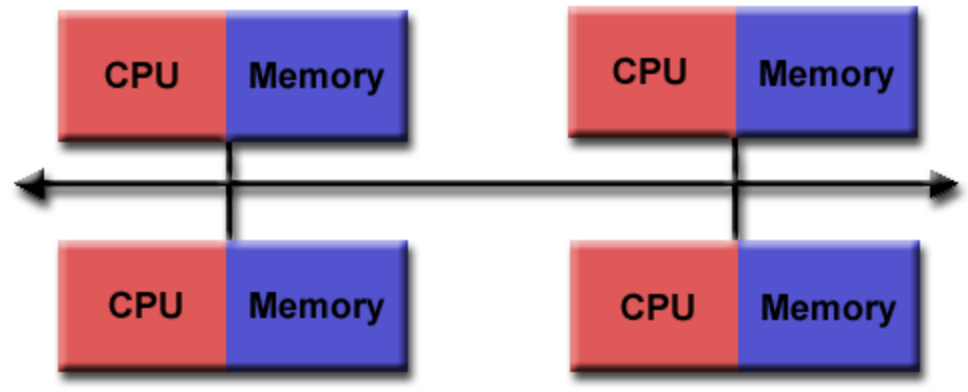
Disadvantages:

- Lack of scalability (geometrical increase of traffic)
- Cost

Memory Architectures

Distributed memory architecture:

- Processors have their own local memory
- Programmers have to ensure that each processors has the necessary data in the local memory
- Each processor operates independently
- Cache Coherency does not apply



Memory Architectures

Advantages of Distributed Memory:

- Memory and processors are scalable
- Cost (commodity hardware)

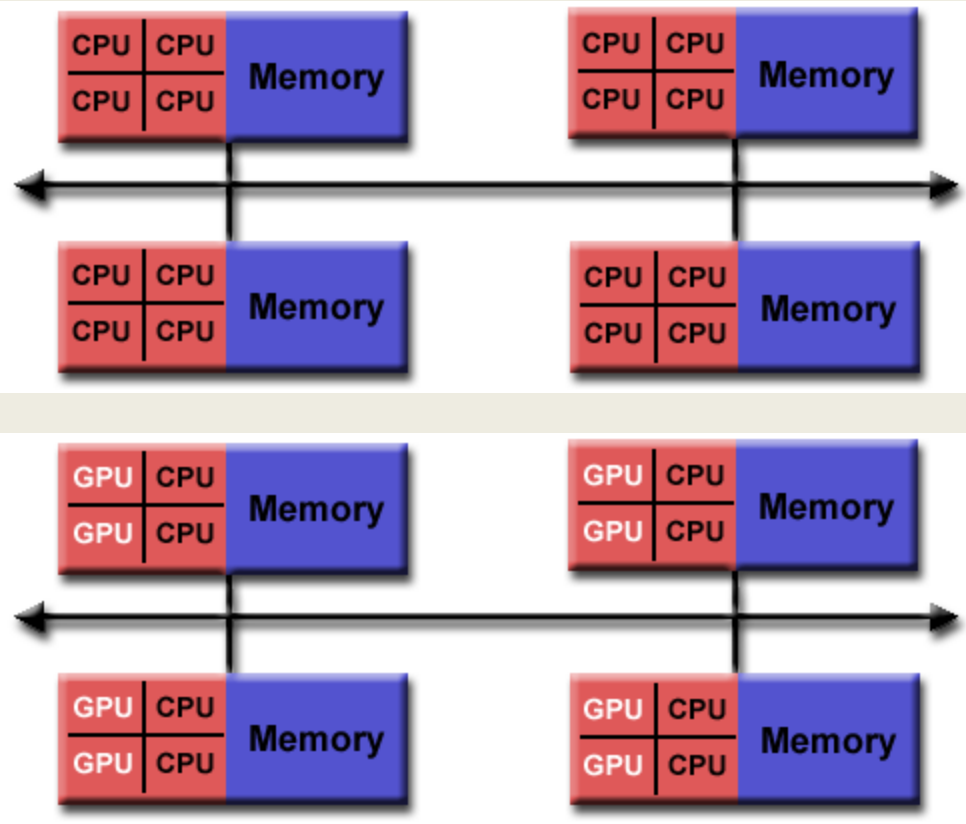
Disadvantages:

- Programmer is responsible for data exchange and communication

Memory Architectures

Hybrid memory architecture:

- Largest computers use hybrid architectures



INTERSECT

Available machines: Mclaren

- SGI Altix 4700
- Shared memory
- 256 Itanium-2
- 1 TB RAM
- SUSE Linux
- At Global Switch

INTERSECT

Available machines: Vayu

- Sun Constellation
- Distributed memory machine
- 12,000 Nehalem CPUs (own 4%)
- 37 TB RAM
- Centos 5.4 Linux
- At NCI/Canberra

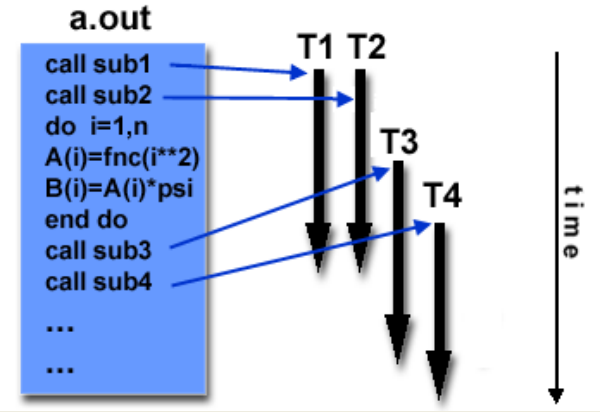
INTERSECT

Parallel Programming Models

- Shared Memory (without threads, native compilers)
- Threads (Posix Threads and OpenMP)
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid
- Single Program Multiple Data
- Multiple Program Multiple Data

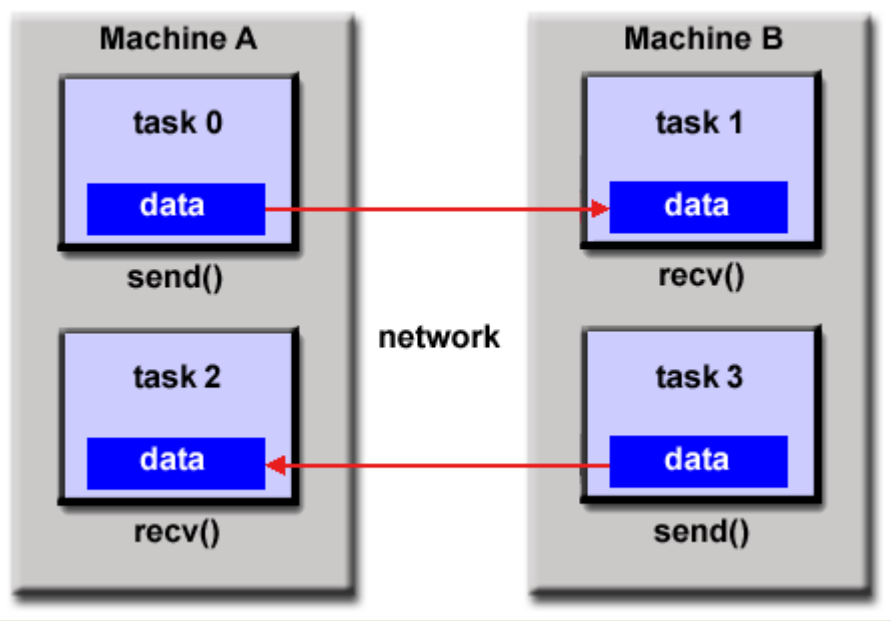
Threads Model

- Type of shared memory model
- Implementations: POSIX (C only) and OpenMP



Message Passing Model

- Type of distributed memory model
- Implementations: Message Passing Interface MPI



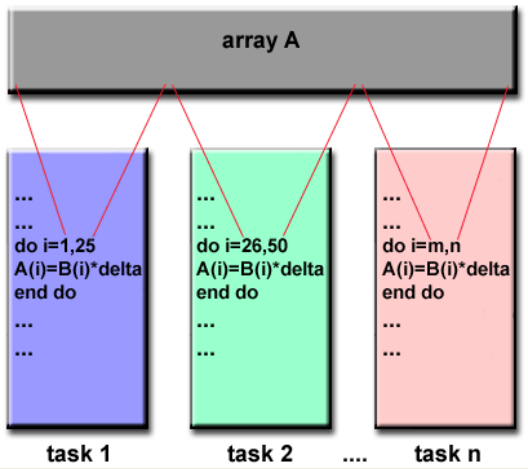
Data Parallel Model

- Implementations: Fortran 90 and 95

  Fortran 77 plus pointers, dynamic memory allocation, array processing as objects, recursive functions, etc.

- High Performance Fortran (HPF)

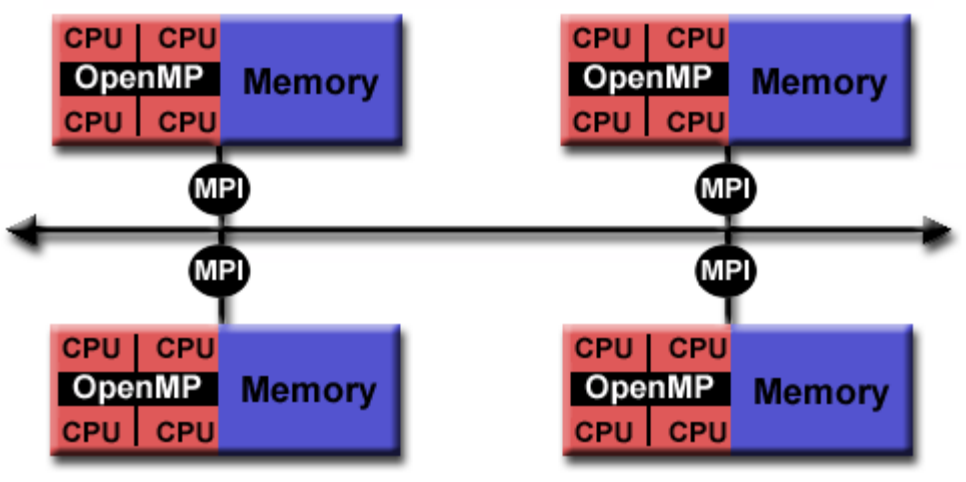
  Fortran 90 plus directives to tell the compiler how to distribute data etc



INTERSECT

Hybrid Model

Message Passing (MPI) plus Threads (OpenMP)



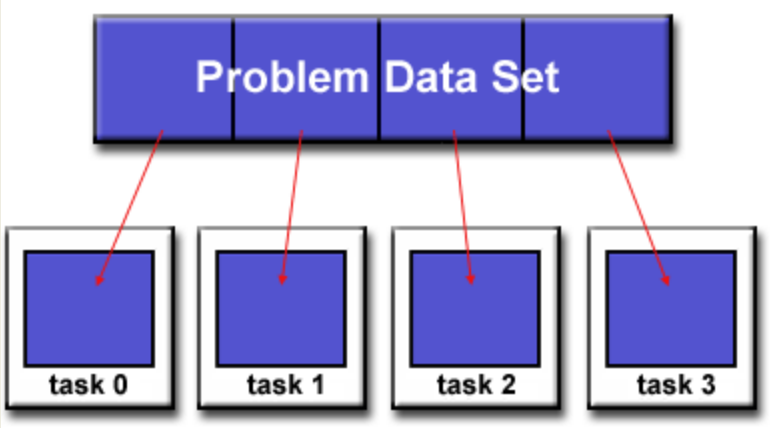
Designing Parallel Programs

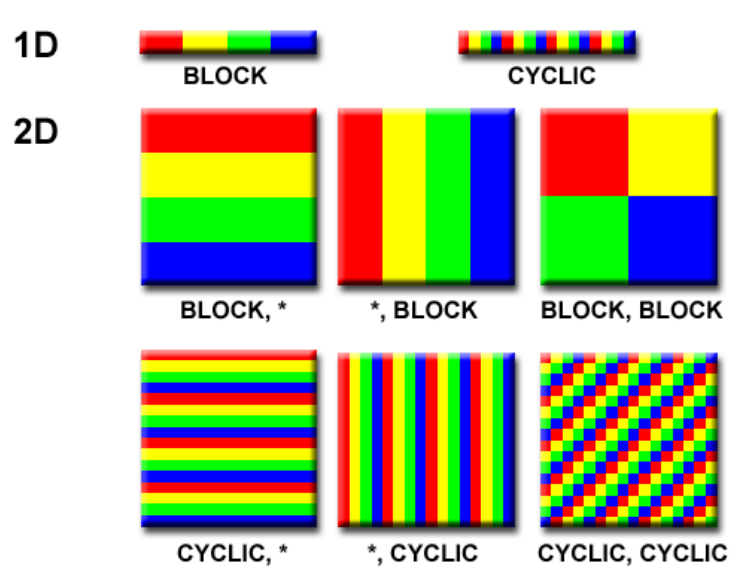
- Determine whether the problem can be parallelized
  F(n)=F(n-1)+F(n-2) Fibonacci non-parallelizable

- Identify hotspots
- Identify bottlenecks
- Identify data dependencies (as F(n))
- Investigate other algorithms

Designing Parallel Programs

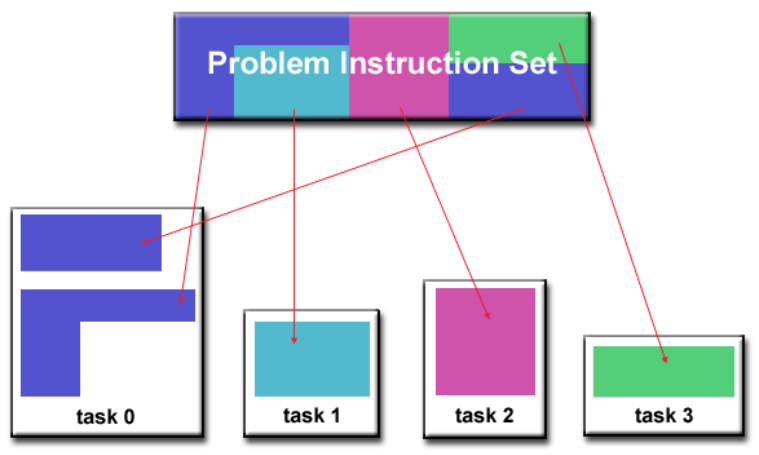
Partitioning: Domain

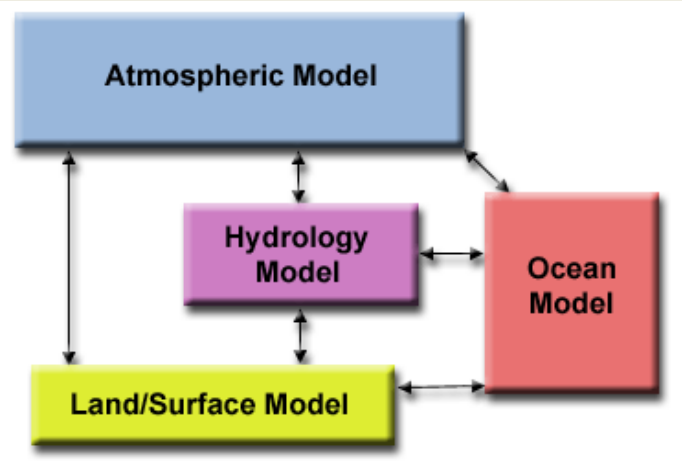




Designing Parallel Programs

Partitioning: Functional





Designing Parallel Programs

Communication:

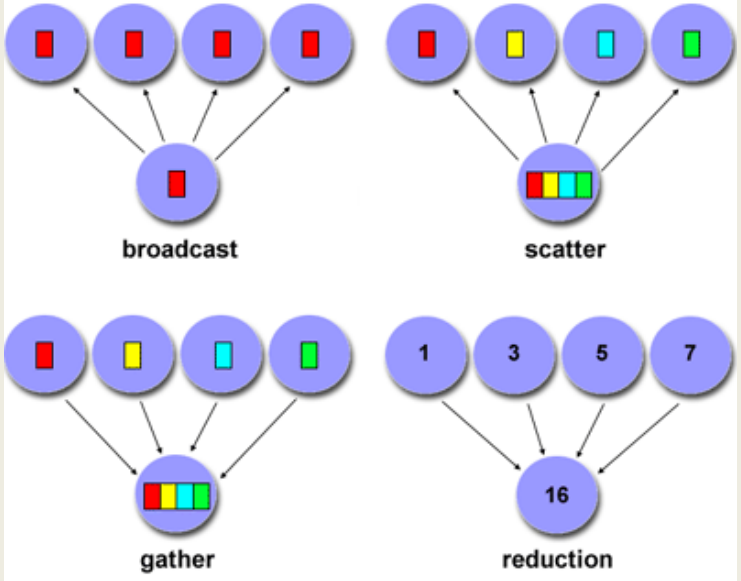
- Most parallel programs need communication (embarrassingly parallel programs do not)

Consider:

- Latency: time it takes to send a 0 byte message from A to B
- Bandwidth: amount of data that can be send in a unit time

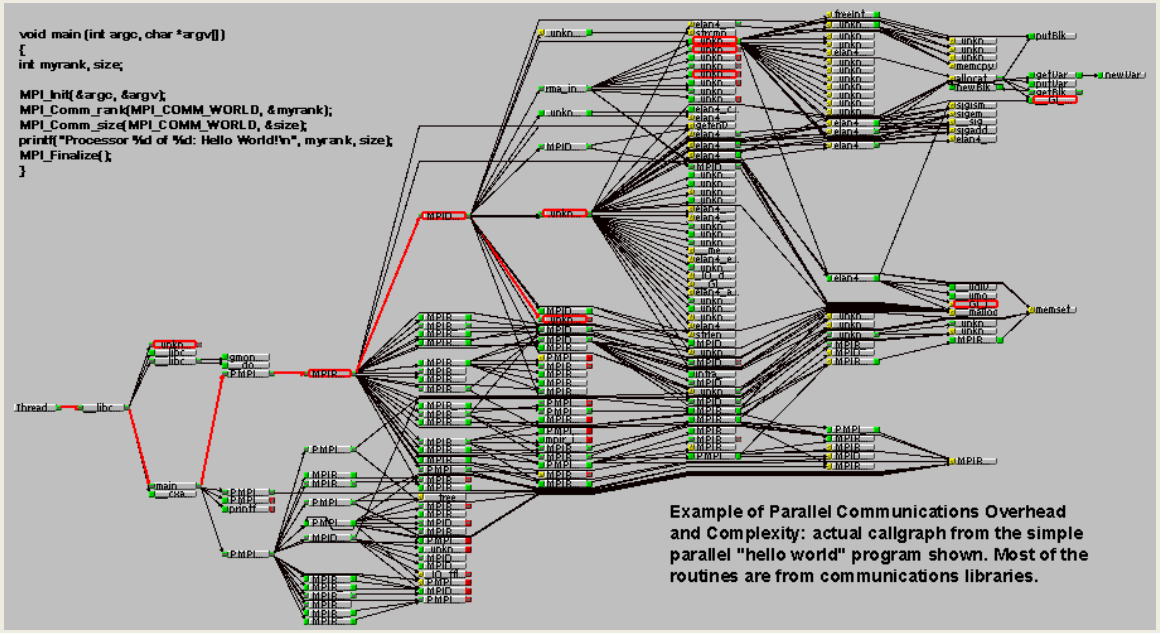
Designing Parallel Programs

Scope of Communication:



Designing Parallel Programs

Overhead and Complexity:



INTERSECT

Designing Parallel Programs

Granularity:

- Fine Grain Parallelism

  Low computation/communication ratio

  Good load balancing

- Coarse Grain Parallelism

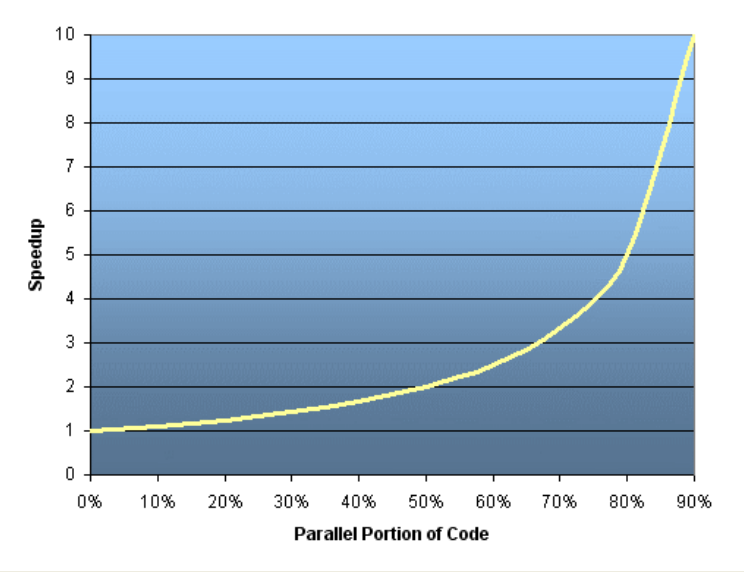
  High computation/communication ratio

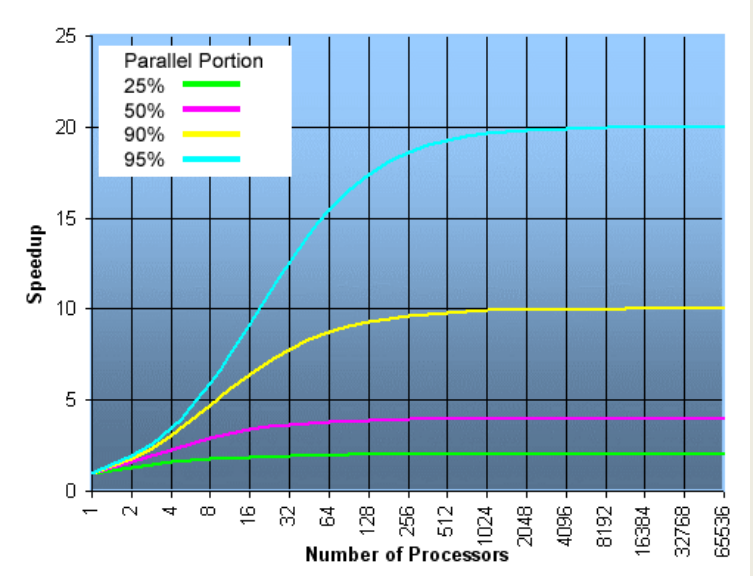
  More difficult load balancing

Designing Parallel Programs

Limits and Costs: Amdahl's Law

Speedup = 1/(1-p)





INTERSECT

Designing Parallel Programs

Many more points to consider:

- Complexity
- Portability
- Resource Requirements
- Scalability
- Etc.

OpenMP

OpenMP runs on a shared memory architecture.

With special programs such as ScaleMP also on a distributed memory architecture.

Application Programming Interface (API).

Not a new language.

It has bindings to C/C++ and Fortran.

OpenMP

Three primary API components:

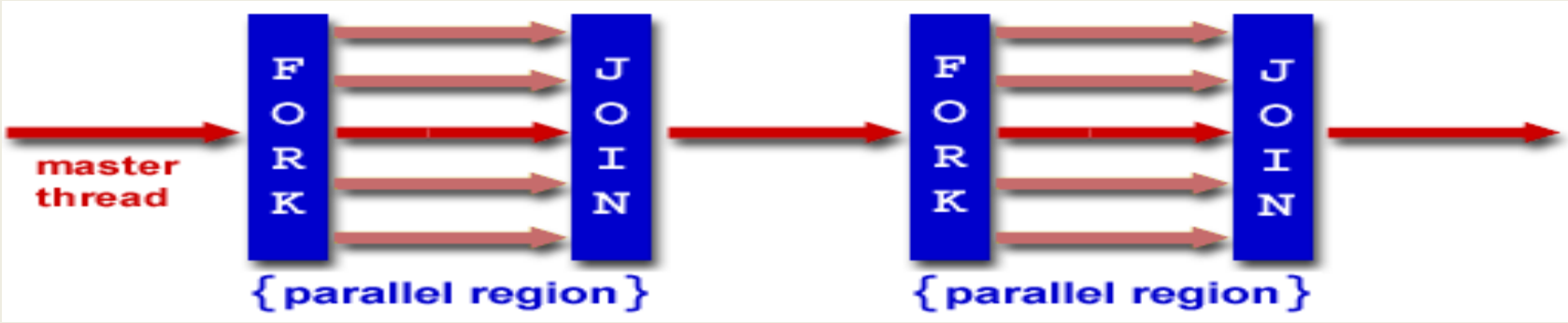
- Compiler directives
- Runtime library routines
- Environment Variables

OpenMP Strong Points:

-Incremental Parallelization
-Portability
-Ease of use
-Standardized

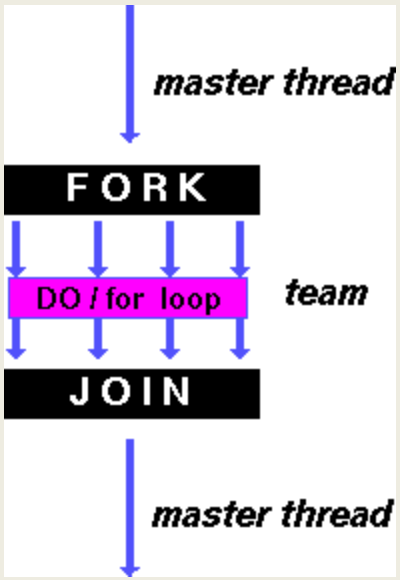
OpenMP

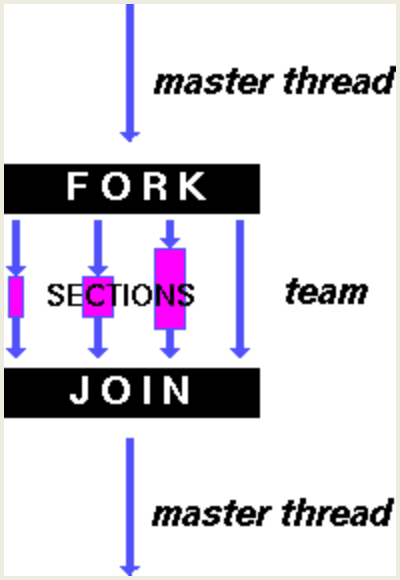
Program Flow:



- Thread based
- Fork-Join Model
- Compiler Directive based
- Dynamic threads

Work-Sharing Constructs





OpenMP

Compiler Directives:
- Fortran: !$OMP (or C$OMP or $OMP)
- C/C++: #pragma omp

Parallel Regions:

```
double A[1000];
omp_set_num_threads (4);
#pragma omp_parallel
{
   int ID = omp_get_thread_num();
   foo (ID, A);
}
printf (“Done\n”);
```

Parallel Region Construct

```
#pragma omp parallel [clause ...]  newline
                    if (scalar_expression)
                    private (list)
                    shared (list)
                    default (shared | none)
                    firstprivate (list)
                    reduction (operator: list)
                    copyin (list)
                    num_threads (integer-expression)
   structured_block

!$OMP PARALLEL [clause ...]
                IF (scalar_logical_expression)
                PRIVATE (list)
                SHARED (list)
                DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
                FIRSTPRIVATE (list)
                REDUCTION (operator: list)
                COPYIN (list)
                NUM_THREADS (scalar-integer-expression)
   block
!$OMP END PARALLEL
```

INTERSECT

Parallel Region: Hello World: C

```
#include <omp.h>
main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

Parallel Region: Hello World: F

```
      PROGRAM HELLO

      INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
     +   OMP_GET_THREAD_NUM

C     Fork a team of threads with each thread having a private TID variable
!$OMP PARALLEL PRIVATE(TID)

C     Obtain and print thread id
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID

C     Only master thread does this
      IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF

C     All threads join master thread and disband
!$OMP END PARALLEL

      END
```

Environment Setup: Modules

Almost no defaults are set. Choose which compiler or program version you want to use.

Commands:

```
module avail
module list
module show
module load name
module unload name
```

Use this for your batch scripts as well!

Compiling Code

module load intel/11.1.073

Intel: icc test.c -o test -openmp
ifort test.f-o test -openmp

PBS script:

```
#!/bin/bash
#PBS -P a40
#PBS -l ncpus=4, walltime=00:01:00
#PBS -wd

export OMP_NUM_THREADS=4
./test
```

Exercise 1: Hello World

Write a hello-world program in C or Fortran. Observe the order of the ranks. Get a feeling to work with the modules.

Hints:
Load the Intel compilers:
module load intel/11.1.073

Compile:
icc hello.c -o hello -openmp
ifort hello.f -o hello -openmp
Set environment:
export OMP_NUM_THREADS=4
Run:
./hello

For/Do Directive: C

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait
   for_loop
```

For/Do Directive: Fortran

```
!$OMP DO [clause ...]
        SCHEDULE (type [,chunk])
        ORDERED
        PRIVATE (list)
        FIRSTPRIVATE (list)
        LASTPRIVATE (list)
        SHARED (list)
        REDUCTION (operator | intrinsic : list)
        COLLAPSE (n)

   do_loop

!$OMP END DO  [ NOWAIT ]
```

Clauses

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team.

STATIC

Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC

Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED

Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.

Clauses

RUNTIME
The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

AUTO
The scheduling decision is delegated to the compiler and/or runtime system.

NO WAIT / nowait: If specified, then threads do not synchronize at the end of the parallel loop.

ORDERED: Specifies that the iterations of the loop must be executed as they would be in a serial program.

COLLAPSE: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Clauses

Private

Private (list)

PRIVATE variables behave as follows:

A new object of the same type is declared once for each thread in the team

All references to the original object are replaced with references to the new object

Variables declared PRIVATE should be assumed to be uninitialized for each thread

Clauses

Shared

Shared (list)

Shared variables behave as follows:

A shared variable exists in only one memory location and all threads can read or write to that address

It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

Clauses

Reduction

Reduction (operator:list)
Reduction (operator|intrinsic:list)

The REDUCTION clause performs a reduction on the variables that appear in its list.

A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Example: Vector Add

Arrays A, B, C, and variable N will be shared by all threads.

Variable I will be private to each thread; each thread will have its own unique copy.

The iterations of the loop will be distributed dynamically in CHUNK sized pieces.

Threads will not synchronize upon completing their individual pieces of work (NOWAIT).

Example: Vector Add: C

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{
int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

Example: Vector Add: F

```
      PROGRAM VEC_ADD_DO

      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=1000)
      PARAMETER (CHUNKSIZE=100)
      REAL A(N), B(N), C(N)

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO
      CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
      DO I = 1, N
        C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
      END
```

Exercise 2: Dot Product

Write a program for a dot product of 2 vectors a and b defined by

$X = \Sigma\ a[i] * b[i]$

Hint:

Use a parallel for construct with the reduction clause.

Exercise 2: Dot Product

Write a program for a dot product of 2 vectors a and b defined by

$X = \Sigma\ a[i] * b[i]$

Hint:

Use a parallel for construct with the reduction clause.

Solution:

```
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
```

```
!$OMP PARALLEL DO REDUCTION(+:SUM)
      DO I = 1, N
        SUM = SUM + (A(I) * B(I))
      ENDDO
```

Exercise 2: Dot Product

Solution (more options specified):

```
#pragma omp parallel for      \
  default(shared) private(i)  \
  schedule(static,chunk)      \
  reduction(+:result)

  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
```

Sections Directive: C

```
#pragma omp sections [clause ...]  newline
                    private (list)
                    firstprivate (list)
                    lastprivate (list)
                    reduction (operator: list)
                    nowait
 {

 #pragma omp section   newline

    structured_block

 #pragma omp section   newline

    structured_block

 }
```

Sections Directive: Fortran

```
!$OMP SECTIONS [clause ...]
            PRIVATE (list)
            FIRSTPRIVATE (list)
            LASTPRIVATE (list)
            REDUCTION (operator | intrinsic : list)

!$OMP  SECTION

   block

!$OMP  SECTION

    block

!$OMP END SECTIONS  [ NOWAIT ]
```

Sections Directive Example: C

```
#include <omp.h>
#define N     1000

main ()
{
int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];
    }  /* end of sections */
  }  /* end of parallel section */
}
```

Sections Directive Example: F

```
      PROGRAM VEC_ADD_SECTIONS

      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N), D(N)

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.5
        B(I) = I + 22.35
      ENDDO

!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION
      DO I = 1, N
        C(I) = A(I) + B(I)
      ENDDO

!$OMP SECTION
      DO I = 1, N
        D(I) = A(I) * B(I)
      ENDDO

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL

      END
```

INTERSECT

Synchronization

THREAD 1:

```
increment(x)
{
   x = x + 1;
}
```

THREAD 1:

```
10  LOAD A, (x address)
20  ADD A, 1
30  STORE A, (x address)
```

THREAD 2:

```
increment(x)
{
   x = x + 1;
}
```

THREAD 2:

```
10  LOAD A, (x address)
20  ADD A, 1
30  STORE A, (x address)
```

Synchronization

One possible execution sequence:

Thread 1 loads the value of x into register A.
Thread 2 loads the value of x into register A.
Thread 1 adds 1 to register A
Thread 2 adds 1 to register A
Thread 1 stores register A at location x
Thread 2 stores register A at location x

The resultant value of x will be 1, not 2 as it should be.

Synchronization: Master

C:

```
#pragma omp master  newline
   structured_block
```

Fortran:

```
!$OMP MASTER
   block
!$OMP END MASTER
```

The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code.

Synchronization: Critical

C:

```
#pragma omp critical [ name ]  newline
   structured_block
```

Fortran:

```
!$OMP CRITICAL [ name ]
   block
!$OMP END CRITICAL [ name ]
```

The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

Example: Critical

```
#include <omp.h>

main()
{
int x=0;

#pragma omp parallel shared(x)
  {

  #pragma omp critical
  x = x + 1;

  }  /* end of parallel section */

}
```

All threads in the team will attempt to execute in parallel, however, because of the CRITICAL construct surrounding the increment of x, only one thread will be able to read/increment/write x at any time.

Synchronization: Barrier

C:
#pragma omp barrier newline

Fortran:
!$OMP BARRIER

The BARRIER directive synchronizes all threads in the team.

When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

Synchronization: Ordered

C:

```
#pragma omp for ordered [clauses...]
   (loop region)
#pragma omp ordered  newline
   structured_block
   (endo of loop region)
```

Fortran:

```
!$OMP DO ORDERED [clauses...]
   (loop region)
!$OMP ORDERED
   (block)
!$OMP END ORDERED
   (end of loop region)
!$OMP END DO
```

Synchronization: Ordered

The ORDERED directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.

Used within a DO / for loop with an ORDERED clause

The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.

Exercise 3: Matrix Multiplication

Write a matrix-matrix multiplication program.

C = A * B

defined by

C(ij) = Sum_k A(ik) * B(kj)

Hint:
Do matrix multiply sharing iterations on outer loop

MPI: Message Passing Interface

-1994. MPI-1 (specification, not strictly a library)
-1996: MPI-2 (addresses some extensions)

Interface for C/C++ and Fortran
Header files:
C: #include <mpi.h>
F: include 'mpif.h'

Compiling:
Intel: icc -lmpi .... (ifort -lmpi …)
Gnu: mpicc … (mpif77, mpif90, mpicxx)
Running:
mpirun -np 4 ./myprog

Reasons for using MPI

Standardization: MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

Portability: There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

Performance: Vendor implementations.

Functionality: Over 115 routines are defined in MPI-1 alone.

Availability: A variety of implementations are available, both vendor and public domain.

Programming Model

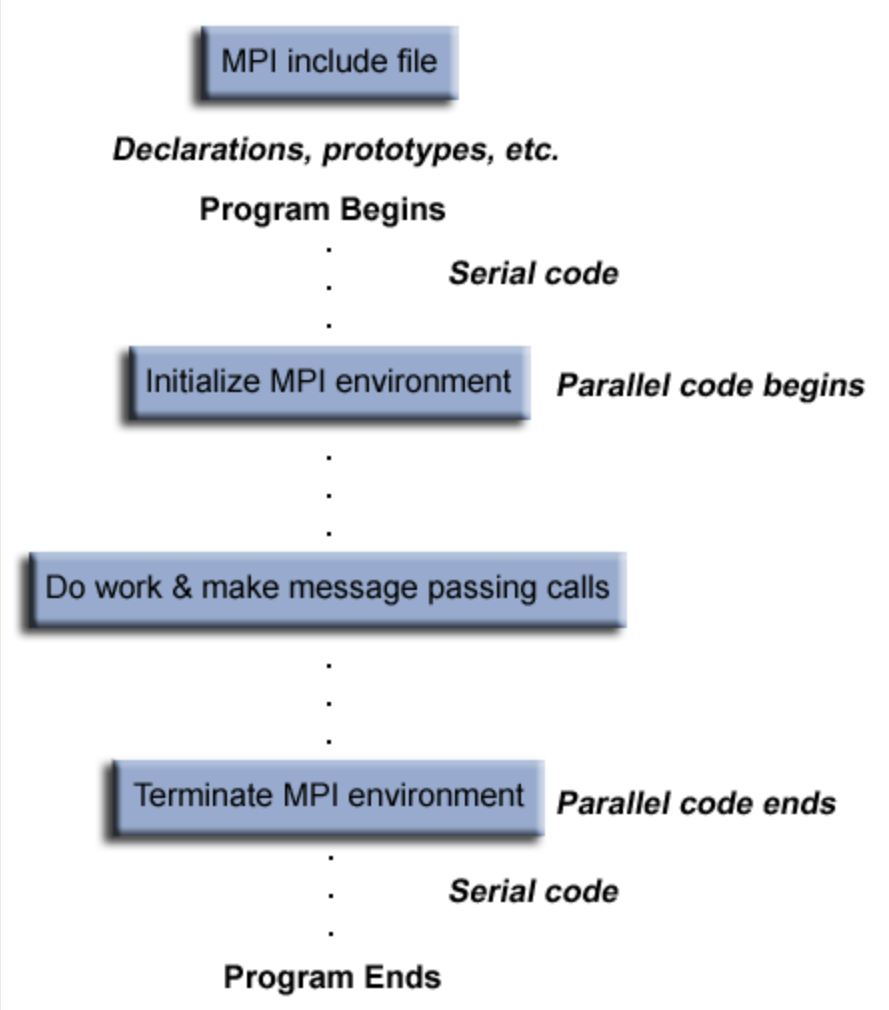
- Distributed programming model. Also data parallel.

- Hardware platforms: distributed, shared, hybrid

- Parallelism is explicit. The programmer is responsible for implementing all parallel constructs.

- The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time. (MPI-2 addresses this issue).

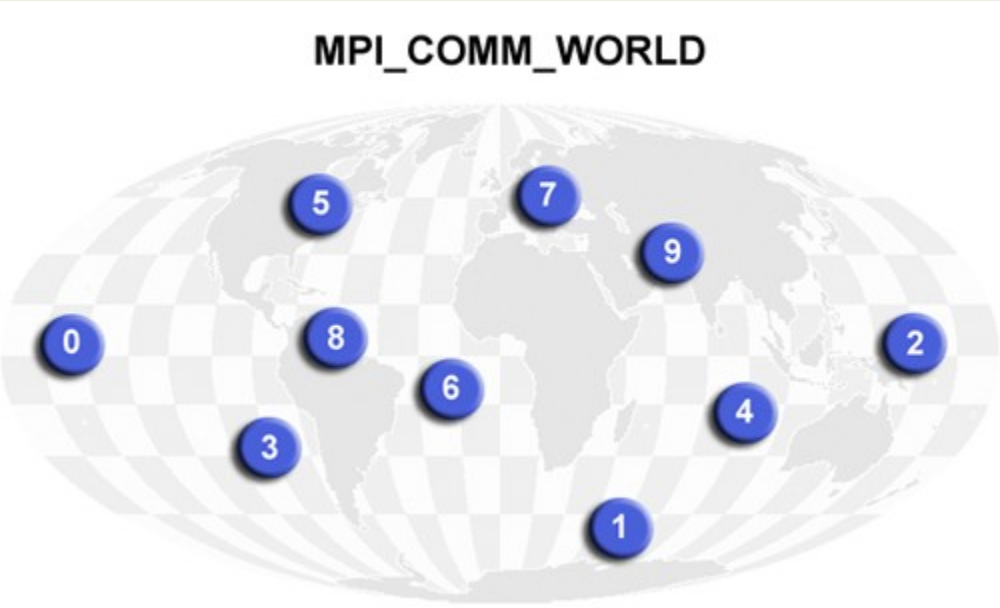
Program Structure



Communicators and Groups

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.

Most MPI routines require you to specify a communicator as an argument.



INTERSECT

Initializing

MPI Init:
MPI_Init (&argc,&argv)
MPI_INIT (ierr)

MPI_Comm_size:
MPI_Comm_size (comm,&size)
MPI_COMM_SIZE (comm,size,ierr)
Determines the number of processes in the group associated with a communicator.

MPI_Comm_rank:
MPI_Comm_rank (comm,&rank)
MPI_COMM_RANK (comm,rank,ierr)
Determines the rank (task ID) of the calling process within the communicator. Value 0...p-1

Initializing

MPI_Abort:
MPI_Abort (comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)
Terminates all MPI processes associated with the communicator.

MPI_Finalize:
MPI_Finalize ()
MPI_FINALIZE (ierr)
Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

Example: C

```
#include <mpi.h>
 #include <stdio.h>

 int main(argc,argv)
 int argc;
 char *argv[]; {
 int  numtasks, rank, rc;

 rc = MPI_Init(&argc,&argv);
 if (rc != MPI_SUCCESS) {
   printf ("Error starting MPI program. Terminating.\n");
   MPI_Abort(MPI_COMM_WORLD, rc);
   }

 MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);

 /*******  do some work *******/

 MPI_Finalize();
 }
```

INTERSECT

Example: F

```
program simple
include 'mpif.h'

 integer numtasks, rank, ierr, rc

 call MPI_INIT(ierr)
 if (ierr .ne. MPI_SUCCESS) then
    print *,'Error starting MPI program. Terminating.'
    call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
 end if

 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
 print *, 'Number of tasks=',numtasks,' My rank=',rank

C ****** do some work ******

 call MPI_FINALIZE(ierr)

 end
```

Point to Point Communication

MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.

Different types of send and receive routines:

- Synchronous send
- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Buffered send
- Combined send/receive
- "Ready" send

Any type of send routine can be paired with any type of receive routine.

Exercise 1: Hello World

Based on the last example write a MPI version of hello world and run it on 4 cores. Each process should print “Hello World” and it's task number.

Hint:
You need the following MPI routines for this exercise:

MPI_Init (&argc,&argv)
MPI_Comm_rank (MPI_COMM_WORLD, &rank)
MPI_Comm_size (MPI_COMM_WORLD, &size)
MPI_Finalize()

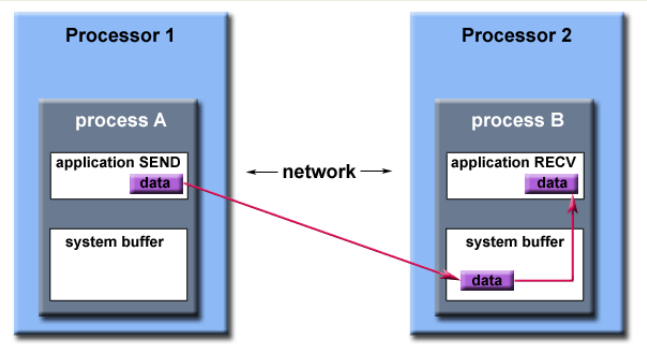
Buffering

In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. The MPI implementation must be able to deal with storing data when the two tasks are out of sync.

Consider the following two cases:

- A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
- Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?

The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a system buffer area is reserved to hold data in transit. For example:



Path of a message buffered at the receiving process

Blocking vs. Non-blocking

Blocking:

A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.

A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.

A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.

A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Non-blocking:

Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.

Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.

It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.

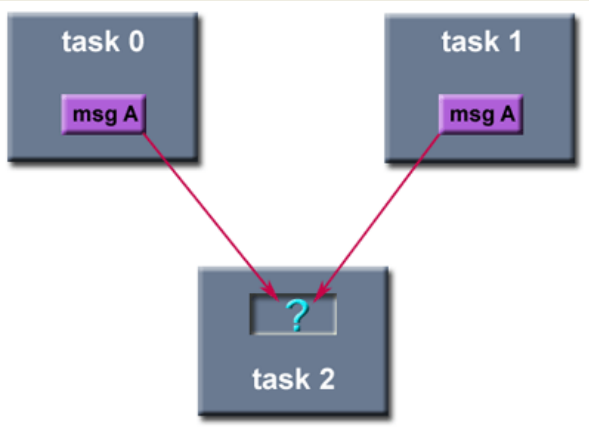
Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

Order and Fairness

Order:
MPI guarantees that messages will not overtake each other.

Fairness:
MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".

Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.



MPI Send / Receive

MPI point-to-point communication routines generally have an argument list that takes one of the following formats:

MPI_Send (&buf,count,datatype,dest,tag,comm)
MPI_SEND (buf,count,datatype,dest,tag,comm,ierr)

Buffer
Program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: &var1

Data Count
Indicates the number of data elements of a particular type to be sent.

Data Type
For reasons of portability, MPI predefines its elementary data types.
MPI_CHAR – signed char
MPI_INT – signed int
MPI_FLOAT – float
MPI_DOUBLE – double

You can also create your own derived data types.

MPI Send / Receive

Destination
An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.

Source
An argument to receive routines that indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.

Tag
Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag. The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

Communicator
Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used.

MPI Send / Receive

Status

For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure MPI_Status (ex. stat.MPI_SOURCE stat.MPI_TAG). In Fortran, it is an integer array of size MPI_STATUS_SIZE (ex. stat(MPI_SOURCE) stat(MPI_TAG)). Additionally, the actual number of bytes received are obtainable from Status via the MPI_Get_count routine.

Request

Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure MPI_Request. In Fortran, it is an integer.

MPI Send / Receive

Blocking sends	MPI_Send(buffer,count,type,dest,tag,comm)
Non-blocking sends	MPI_Isend(buffer,count,type,dest,tag,comm,request)
Blocking receive	MPI_Recv(buffer,count,type,source,tag,comm,status)
Non-blocking receive	MPI_Irecv(buffer,count,type,source,tag,comm,request)

MPI_Send:Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.
MPI_Send (&buf,count,datatype,dest,tag,comm)
MPI_SEND (buf,count,datatype,dest,tag,comm,ierr)

MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
MPI_RECV (buf,count,datatype,source,tag,comm,status,ierr)

Synchronous blocking send:Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.
MPI_Ssend (&buf,count,datatype,dest,tag,comm)
MPI_SSEND (buf,count,datatype,dest,tag,comm,ierr)

Buffered blocking send:permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered. Insulates against the problems associated with insufficient system buffer space.
MPI_Bsend (&buf,count,datatype,dest,tag,comm)
MPI_BSEND (buf,count,datatype,dest,tag,comm,ierr)

Blocking Msg Passing Example: C

```
#include <mpi.h>
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize();
}
```

INTERSECT

Blocking Msg Passing Example: F

```
      program ping
      include 'mpif.h'

      integer numtasks, rank, dest, source, count, tag, ierr
      integer stat(MPI_STATUS_SIZE)
      character inmsg, outmsg
      outmsg = 'x'
      tag = 1

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      if (rank .eq. 0) then
        dest = 1
        source = 1
        call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
     &           MPI_COMM_WORLD, ierr)
        call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
     &           MPI_COMM_WORLD, stat, ierr)

      else if (rank .eq. 1) then
        dest = 0
        source = 0
        call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
     &       MPI_COMM_WORLD, stat, err)
        call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
     &       MPI_COMM_WORLD, err)
      endif

      call MPI_GET_COUNT(stat, MPI_CHARACTER, count, ierr)
      print *, 'Task ',rank,': Received', count, 'char(s) from task',
     &         stat(MPI_SOURCE), 'with tag',stat(MPI_TAG)
      call MPI_FINALIZE(ierr)
      end
```

INTERSECT

Exercise 2: Ping

Write a MPI program which sends a message to another process which receives it and sends it back. For this you need 2 processes. Test whether the program was invoked with more than 2 processes and display a warning that the program will only use 2 processes.

Run the program with:

mpirun -np 2 ./ping

Non-Blocking Msg Passing

MPI_Isend
Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.

MPI_Irecv
Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

MPI_Issend
Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message.

MPI_Ibsend
Non-blocking buffered send. Similar to MPI_Bsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Must be used with the MPI_Buffer_attach routine.

MPI_Irsend
Non-blocking ready send. Similar to MPI_Rsend() except MPI_Wait() or MPI_Test() indicates when the destination process has received the message. Should only be used if the programmer is certain that the matching receive has already been posted.

Non-Blocking Msg Passing: C

```
#include <mpi.h>                        /* Nearest neighbor exchange in ring topology */
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[2];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    {  do some work  }

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
}
```

Non-Blocking Msg Passing: F

```
  program ringtopo
  include 'mpif.h'
  integer numtasks, rank, next, prev, buf(2), tag1, tag2, ierr
  integer stats(MPI_STATUS_SIZE,2), reqs(4)
  tag1 = 1
  tag2 = 2

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

  prev = rank - 1
  next = rank + 1
  if (rank .eq. 0) then
    prev = numtasks - 1
  endif
  if (rank .eq. numtasks - 1) then
    next = 0
  endif

  call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1,
 &     MPI_COMM_WORLD, reqs(1), ierr)
  call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2,
 &     MPI_COMM_WORLD, reqs(2), ierr)

  call MPI_ISEND(rank, 1, MPI_INTEGER, prev, tag2,
 &     MPI_COMM_WORLD, reqs(3), ierr)
  call MPI_ISEND(rank, 1, MPI_INTEGER, next, tag1,
 &     MPI_COMM_WORLD, reqs(4), ierr)

C       do some work

  call MPI_WAITALL(4, reqs, stats, ierr);
  call MPI_FINALIZE(ierr)
  end
```

INTERSECT

Collective Communication Routines

MPI_Barrier
Creates a barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call.
MPI_Barrier (comm)
MPI_BARRIER (comm,ierr)

MPI_Bcast
Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.
MPI_Bcast (&buffer,count,datatype,root,comm)
MPI_BCAST (buffer,count,datatype,root,comm,ierr)

MPI_Scatter
Distributes distinct messages from a single source task to each task in the group.
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,
..... recvcnt,recvtype,root,comm)
MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf,
..... recvcnt,recvtype,root,comm,ierr)

MPI_Gather
Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,
..... recvcount,recvtype,root,comm)
MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf,
..... recvcount,recvtype,root,comm,ierr)

MPI_Reduce
Applies a reduction operation on all tasks in the group and places the result in one task.
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)

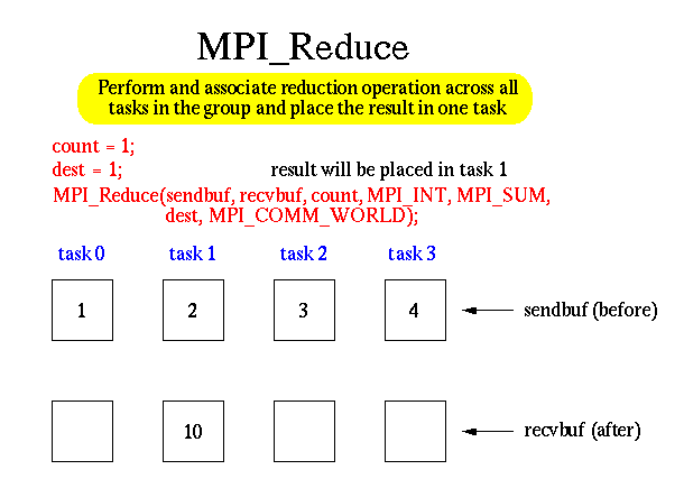
Collective Communication Routines

MPI_Reduce
Applies a reduction operation on all tasks in the group and places the result in one task.
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)

Predefined operations are:

MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
etc

Users can also define their own reduction functions by using the MPI_Op_create routine.



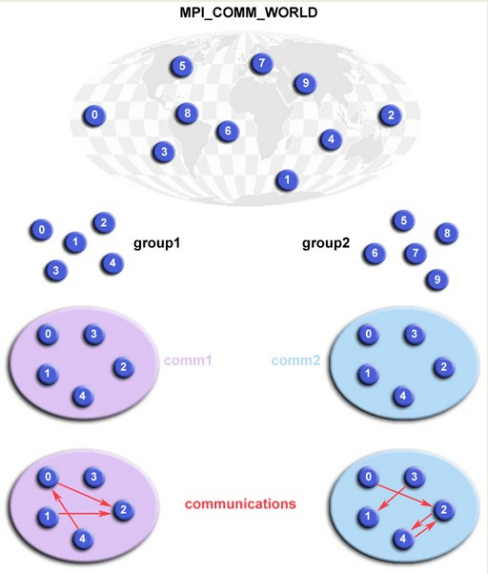
INTERSECT

Communicator Groups

A group is an ordered set of processes. Each process in a group is associated with a unique integer rank.

Allow you to organize tasks, based upon function, into task groups.
Provide basis for implementing user defined virtual topologies
Provide for safe communications

Groups/communicators are dynamic - they can be created and destroyed during program execution.
MPI has over 40 routines related to groups, communicators, and virtual topologies.



INTERSECT

Exercise 3: Pi

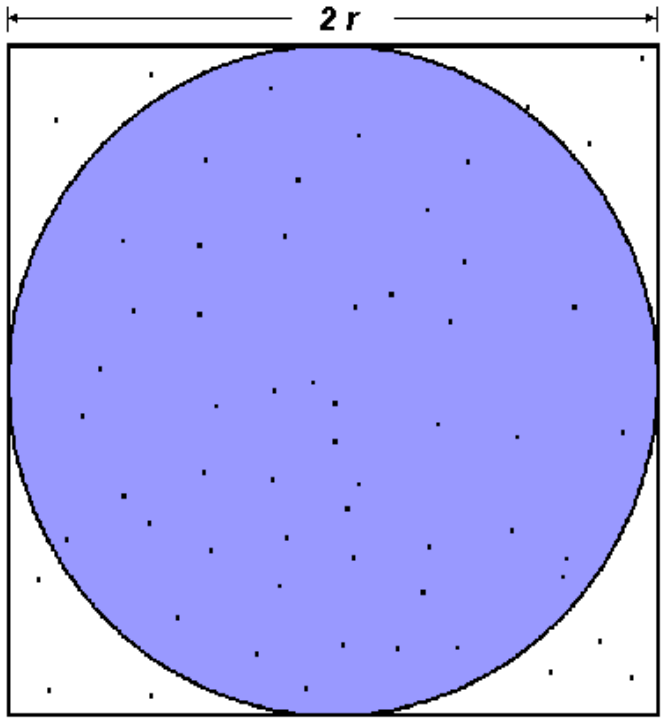
Write a MPI program which calculates pi using a Monte Carlo method.

Hints:
Serial pseudo code:

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Exercise 3: Pi (continued)

Parallel pseudo code:

```
npoints = 10000
circle_count = 0
p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER
  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
Endif
```

Hint: Use rc = MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE, MPI_SUM,
MASTER, MPI_COMM_WORLD);

Virtual Topologies

In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape" such as Graphs or Cartesian Grids.

They are virtual: no relation to the underlying hardware.

They are build on communicators and groups.

Example: A simplified mapping of processes into a Cartesian virtual topology:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

MPI-2

Dynamic Processes - extensions that remove the static process model of MPI. Provides routines to create new processes.

One-Sided Communications - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.

Extended Collective Operations - allows for non-blocking collective operations and application of collective operations to inter-communicators

External Interfaces - defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers.

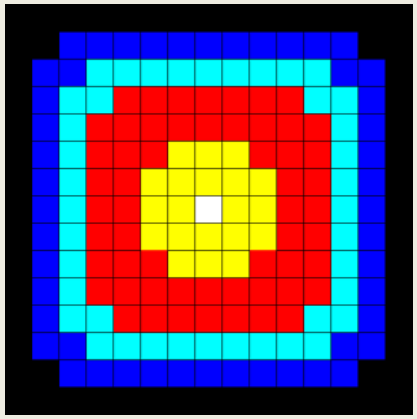
Additional Language Bindings - describes C++ bindings and discusses Fortran-90 issues.

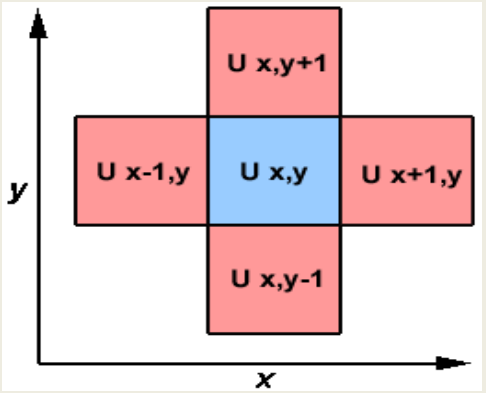
Parallel I/O - describes MPI support for parallel I/O.

Homework

Write a MPI program to solve the Heat equation.

$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) + C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$





A serial program would look like:

```
do iy = 2, ny - 1
do ix = 2, nx - 1
  u2(ix, iy) =
    u1(ix, iy)  +
      cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
      cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
end do
end do
```

INTERSECT

Homework (cont)

Parallel Code as an SPMD model:

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray
  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do t = 1, nsteps
    update time
    send neighbors my border info
    receive from neighbors their border info

    update my portion of solution array

  end do

  send MASTER results

endif
```

Conclusions / Implementations

Advantages / Disadvantages:

- Can scale very nicely up to thousands of cores
- Difficult to program and to debug
- The whole program have to be designed for MPI

Different MPI implementations:

- MPICH, MPICH2
- MVAPICH
- OpenMPI
- SGI MPT

Don't mix MPI/compiler combinations!

Getting Help

Website: Intersect and NCI

Email: hpc_support@intersect.org.au
help@nf.nci.org.au

Sample scripts on Mclaren: /usr/local/doc

Courses:
-Introduction into Linux
-Introduction to HPC@Intersect
-Parallel Programming: OpenMP and MPI
-Programming in Fortran 90 (planned)

INTERSECT

Acknowledgment

Blaise Barney, Lawrence Livermore National Laboratory
www.llnl.gov

INTERSECT

Thank you for your attention!

INTERSECT