Introduction to HPC Approaches

Dr. Cevat Şener

Why HPC?

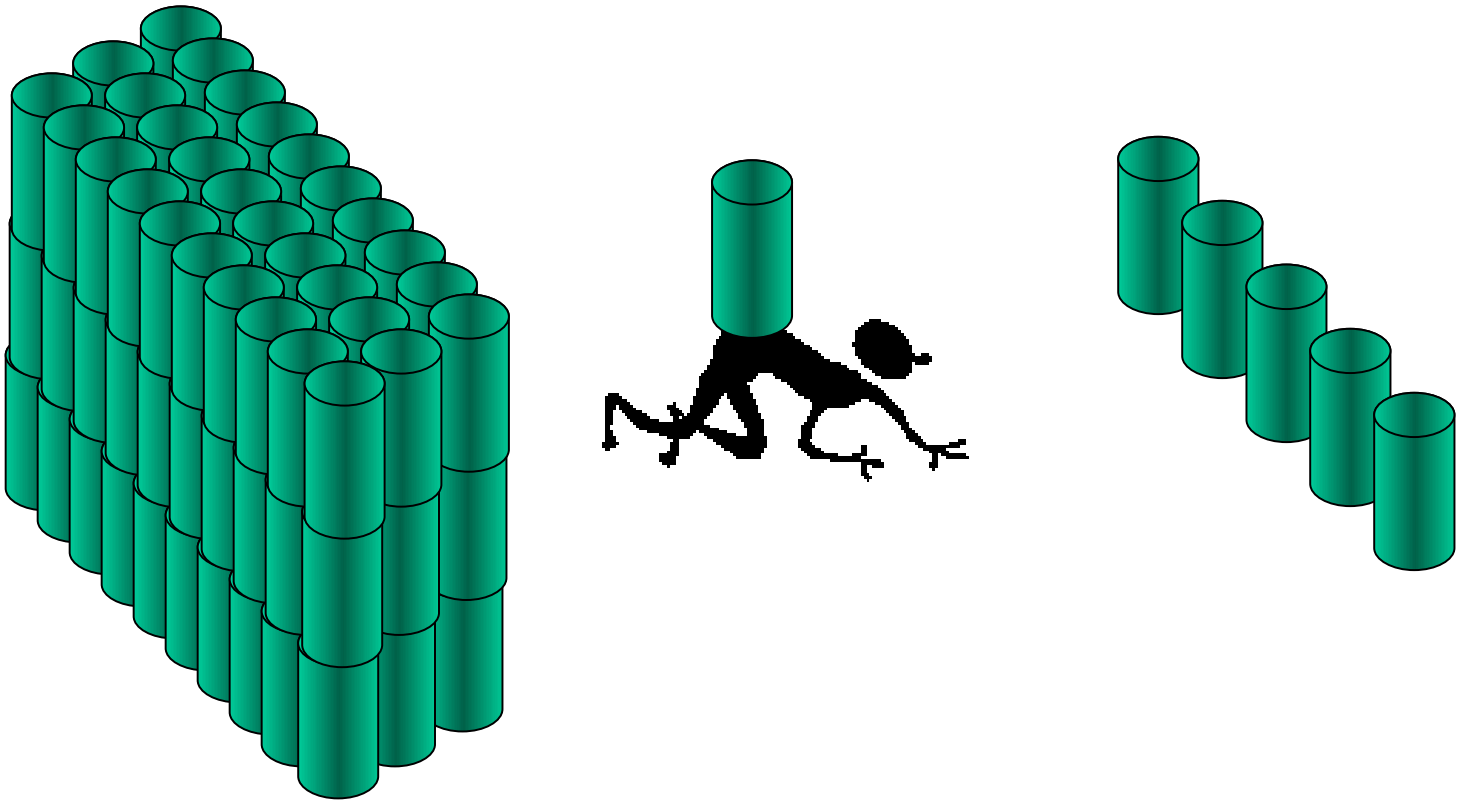
- Many important problems could not be solved yet even with the fastest computers available
- Faster computers enable the formulation of more interesting questions
- When a problem is solved, researchers find biggers to tackle

Grand Challenge Problem Areas

- Wheather forecasting
- Economic modeling
- Computer-aided design
- Drug design
- Exploring the origins of universe
- Searching for extra-terrestial life
- Computer vision
- Bio-informatics
- …

Sequential Processing

- Single CPU

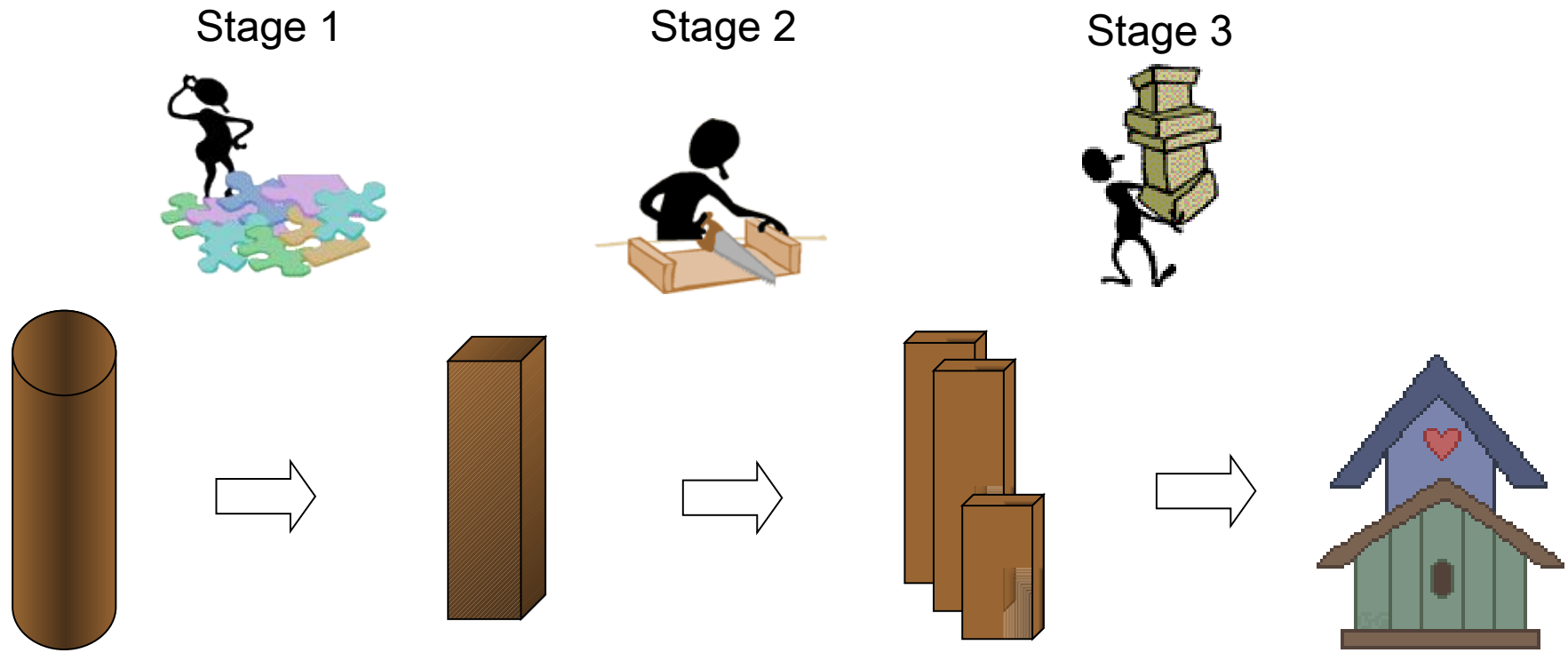


Parallel Processing will Help

- Modes of parallelism to achieve:
  - Pipeline parallelism
  - Data parallelism
  - Control parallelism

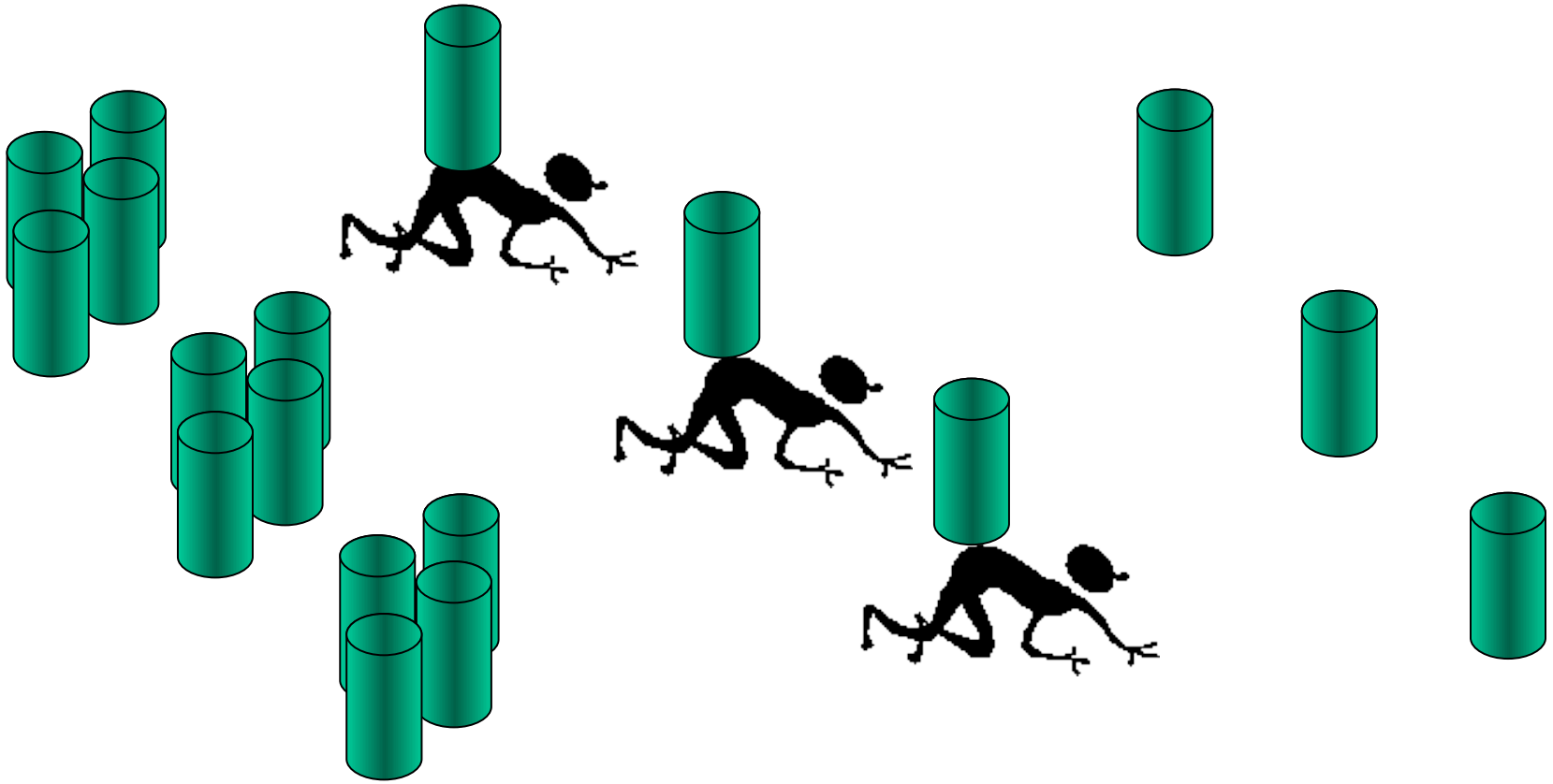
Pipeline Parallelism

- A number of steps called **segments** or **stages**
- The output of one segment is the input of other segment



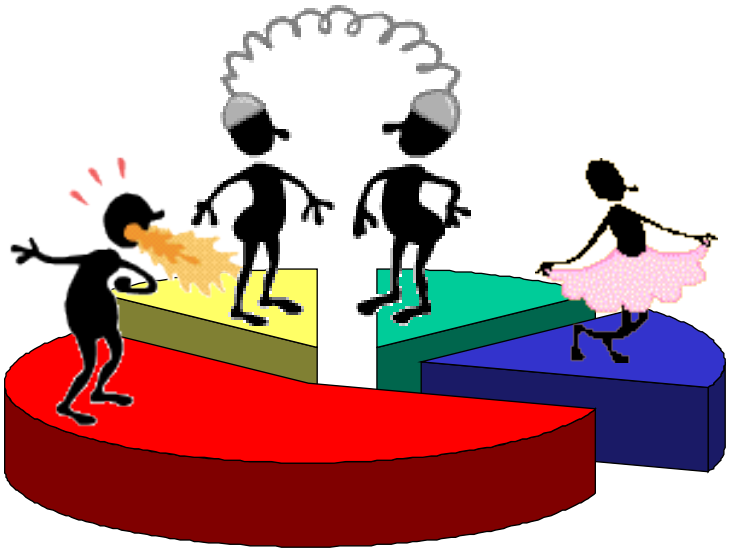
Data Parallelism

- Applying the same operation simultaneously to elements of a data set



Control Parallelism

- Applying different operations to different data elements simultaneously



What to Achieve: Speed-up

- It refers to how much a parallel algorithm is faster than a corresponding sequential algorithm

  $S_p = T_1 / T_n$

  where

  p is the number of processors

  T_1 is the execution time of the sequential algorithm

  T_p is the execution time of the parallel algorithm with p processors

- Ideal case when $S_p = p$
- However, it may not possible to achieve due to overheads, e.g. Communication, synchronization etc.
- Called *scalable* if its speed-up remains acceptable when p gets large

Parallel Computers: Scale & Architecture

- Single chip
- Single system (node)
- Single location (cabinet, room, organization)
- Geographically distributed

Single Chip

- Pipelined
  - Instructions are divided into a number of steps (segments, stages)
  - At the same time, several instructions can be loaded in the machine and be executed in different steps
- Multi-Core
  - aka Chip-level Multi-Processor (CMP)
  - Current processor trend
  - Supports multi-threading successfully and efficiently in hardware

Single System

- Multiprocessors
- Consists of many fully programmable processors each capable of executing its own program
- Shared Address Space Architecture
- Types:
  - SMP (Symmetric Multiprocessor)
    - » A small number of microprocessors connected by a high-speed bus or crossbar switch
  - DSM (Distributed Shared Memory)
    - » The memory is physically distributed among nodes.
  - PVP (Parallel Vector Processor)
    - » A small number of proprietary vector processors connected by a high-bandwidth crossbar switch

Single Location

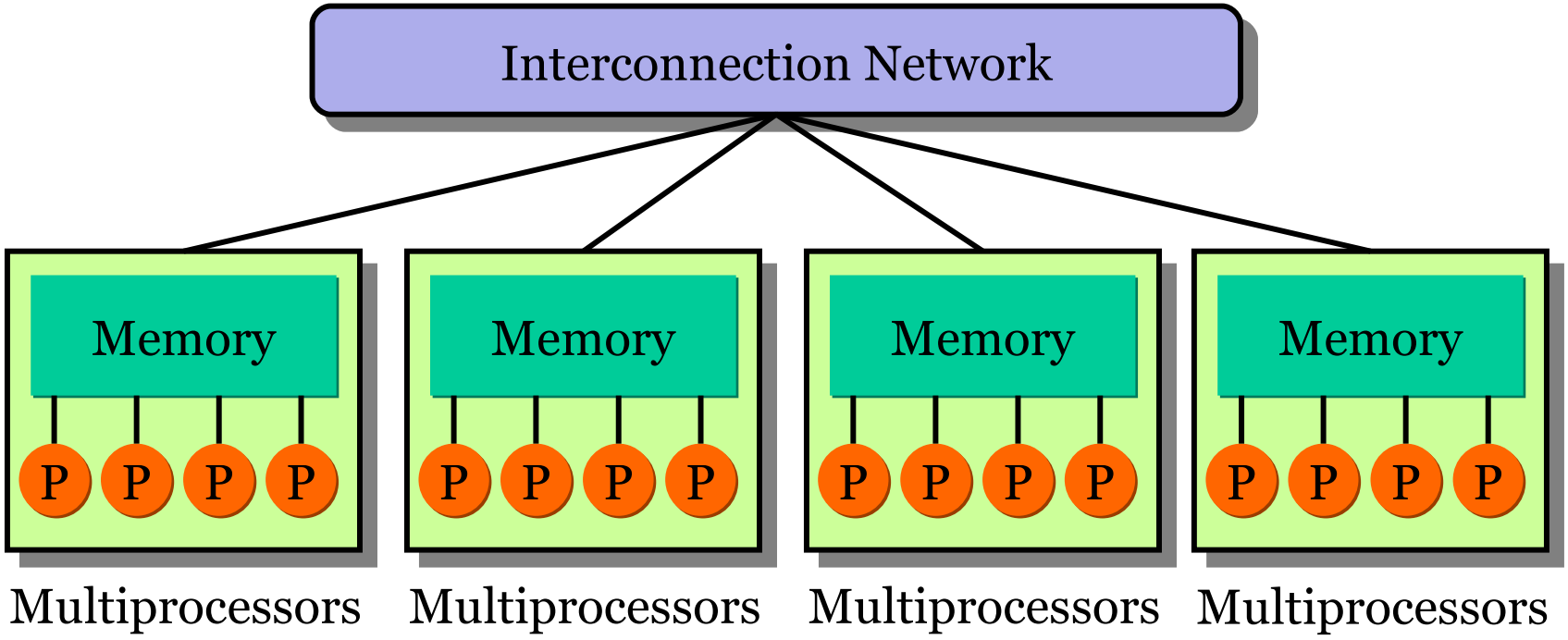
- Multicomputers
- Consists of many processors with their own memory
- No shared memory
- Processors interact via message passing → loosely coupled system
- Types:
  - MPP (Massively Parallel Processing)
    - » Total number of processors > 1000; e.g. BlueGene
  - Cluster
    - » Each node in system has less than 16 processors.
  - Constellation
    - » Each node in system has more than 16 processors

Geographically Distributed

- Grid(s)
- Interconnection of geographically distributed clusters or computers
- Very large capacity

Current Trend in Supercomputers

- Clusters of SMPs,
  - approaching to Constellations



Parallel Programming Models

- Parallel programming models are categorized as
  - Implicit parallelism vs. Explicit parallelism
  - Data-parallel model vs. Control-parallelism
  - Message-passing model vs. Shared-variable model
  - Fine-grained vs. Coarse-grained parallelism

- Models are not and should not be specific to hardware. All should be able to be implemented on any hardware, theoretically.
  - e.g. shared-variable programming on distributed memory hardware, or message-passing on shared memory

Implicit Parallelism

- The compiler and the run-time support system automatically exploit the parallelism from the sequential-like program written by users

Implicit Parallelism

- Ways to implement implicit parallelism
  - Parallelizing (restructuring) compilers
    - » Compiler performs data-dependence and control-dependence analysis on a sequential program's source code
    - » Uses transformation techniques to convert sequential code into parallel
  - User directions
    - » User helps the compiler by providing additional information to guide the parallelization process or by inserting compiler directives in the source code
    - » User is responsible for ensuring that the code is correct after parallelization
  - Run-time parallelization
    - » Involves both the compiler and the run-time system

Explicit Parallelism

- The representation of concurrent computations by means of primitives in the form of special-purpose function calls
- The programmer should explicitly use these primitives to achieve parallelism
- The absolute programmer control over the parallel execution

Data-Parallel Model

- The same instruction or program segment executes over different data sets simultaneously
- Massive parallelism is exploited at data set level
- Has a single thread of control
- Has a global naming space
- Applies loosely synchronous operation
- Parameter Study also falls into this category

Control-Parallelism

- aka Task parallelism or Function parallelism
- A form of parallelization of computer code across multiple processors in parallel computing environments
- Focuses on distributing execution of tasks (processes or threads) across different parallel computing nodes (processors, cores)

Message-Passing Model

- Multithreading: program consists of multiple processes
  - Each process has its own thread of control
  - Both control parallelism (MPMD) and data parallelism (SPMD) are supported
- Asynchronous Parallelism
  - All process execute asynchronously
  - Must use special operation to synchronize processes
- Multiple Address Spaces
  - Data variables in one process is invisible to the others
  - Processes interact by sending/receiving messages
- Explicit Interactions
  - Programmer must resolve all the interaction issues: data mapping, communication, synchronization and aggregation
- Explicit Allocation
  - Both workload and data are explicitly allocated to the process by the user

Shared-Variable Model

- Has a single address space
- Has multithreading and asynchronous model
- Data reside in a single, shared address space, thus does not have to be explicitly allocated
- Workload can be implicitly or explicitly allocated
- Communication is done implicitly
  - Through reading and writing shared variables
- Synchronization is explicit

Fine-Grained Parallelism

- Generraly, every loop has a chance of parallelism
- In this model, many of these loops are parallelized
- No need to know the details of the source or the algorithm
- Needs too frequent data-distribution, sychronization and result-collection processes thrughout the execution
  - Not that suitable over message-passing platforms

Coarse-Grained Parallelism

- Wider loops or larger modules are made parallel
- Less often data-distribution, sychronization and result-collection processes
- Source code and the algorithm need to be well-understood

Comments on Models

- Implicit parallelism
  - Easy to use
  - Can reuse existing sequential programs
  - Programs are portable among different architectures
- Data parallelism
  - Programs are always determine and free of deadlocks/livelocks
  - Difficult to realize some loosely sync. program

Comments on Models

- Message-passing model
  - More flexible than the data-parallel model
  - Lacks support for the work pool paradigm and applications that need to manage a global data structure
  - Widely-accepted
  - Exploit large-grain parallelism and can be executed on machines with native shared-variable model
- Shared-variable model
  - No widely-accepted standard → programs have low portability
  - Programs are more difficult to debug than message-passing programs

Examples

- MPI
- POSIX Threads
- HPF
- OpenMP
- Unified Parallel C
- JEE with Web Services
- MatLab with Distributed Toolbox
- Workflows
- Hybrids

MPI (Message Passing Interface)

- Exploits explicit and message-passing parallelism
- Most suitable to achieve control-parallelism,
  but not that successful on fine-grained parallelism
- A standard portable message-passing library definition developed in 1993 by a group of parallel computer vendors, software writers, and application scientists.
- Many implememantations exist, like MPICH, OpenMPI etc.
- Available to both Fortran and C programs
- Available on a wide variety of parallel machines.

MPI Example

```
#include <mpi.h>
#define N 1000000
main(){
  double pi, sums, v, w=1.0/N;
  int i, mid, nth;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mid);
  MPI_Comm_size(MPI_COMM_WORLD, &nth);
  for(i=mid; i<N; i+= nth) {
    v = (i+0.5) * w;
    sums += 4.0/(1.0+v*v);
  }
  MPI_Reduce(&sums,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  pi *= w;
  if(mid == 0) printf("pi = %f\n",pi);
  MPI_Finalize();
}
```

POSIX Threads

- aka pthreads
- Belongs to explicit and shared-variable parallelism models
- Specified by the IEEE POSIX 1003.1c standard
- Requires significant programmer attention to detail
- Most suitable to achieve control-parallelism

POSIX Threads Example

```
#include <pthread.h>
#define MAX_THREADS 128
#define N 1000000
static int N_Threads = 0;
static double SUM[MAX_THREADS];
static double w;
main(){
  int i; double pi;
  pthread_t thr[MAX_THREADS];
  int args[MAX_THREADS];
  N_Threads = atoi(getenv("N_Threads"));
  w = 1.0/(N*N_Threads);
  pthread_setconcurrency(N_Threads);
  for(i=0; i<N_Threads-1; i++) {
    args[i] = i;
    if(pthread_create(&thr[i],(void*(*))PI,&args[i])){
      perror("Thread create"); exit(1);
    }
  }
  PI(&i); /* let main do part of the calculations */
  pi = SUM[i];
  for(i=0; i<N_Threads-1; i++) {
    pthread_join(thr[i], NULL);
    pi += SUM[i];
  }
  printf("pi = %f\n", pi*w);
}
```

```
void *PI(void *myid){
  int i; mid = *(int *)myid;
  double t = N*mid + 0.5;
  double ss = 0.0;
  for(i=0; i<N; i++) {
    v = (i+t)*w;
    ss += 4.0/(1.0+v*v);
  }
  SUM[mid] = ss;
  return NULL;
}
```

High Performance Fortran (HPF)

- aka Fortran 95
- An extension of Fortran 90 with constructs that support parallel computing, published by the High Performance Fortran Forum (HPFF)
- Assumes shared-variable model, and employs a parallelizing compiler
- Most of the users and vendors have moved to OpenMP

HPF Example

```
PROGRAM piprog
INTEGER, PARAMETER:: N=1000000
REAL (KIND=8):: pi, w=1.0/N

pi = SUM( (/ (4.0*w/(1.0+((i+0.5)*w)**2), i=1,N) /) )
PRINT *, pi

END
```

OpenMP (Open Multi-Processing)

- Employs semi-implicit with compiler directives* and shared-variable parallelism
- An implementation of multithreading
- Consists of a set of compiler directives, together with library routines and environment variables
- Currently only runs efficiently on shared-memory multiprocessor / multicore platforms
- In C/C++ and Fortran on many architectures
- Most suitable for implementing data parallel paradigm

(*) in some references, it is accepted as “explicit” due to the need for inserting these directives

OpenMP Example

```
#define N 1000000
main(){
  double pi, sums, v, w = 1.0/N;
  int i;
#pragma omp parallel for private(i,v) reduction(+:sums)
  for(i=0; i<N; i++) {
    v = (i+0.5)*w;
    sums += 4.0/(1.0+v*v);
  }
  pi = sums*w;
  printf("pi = %f\n",pi);
}
```

Unified Parallel C (UPC)

- An extension of the C programming language designed for high-performance computing
- The programmer is presented with a single shared, partitioned address space, where
  - variables may be directly read and written by any processor,
  - but each variable is physically associated with a single processor
- UPC uses data-parallel model of computation
- Supports explicit parallelism and shared-variable models

UPC Example

```
#include <upc_relaxed.h>
#define N 4
#define P 4
#define M 4
shared [N*P/THREADS] int a[N][P],c[N][M];
shared [M/THREADS] int b[P][M];
void main() {
  int i,j,l;
  upc_forall(i=0;i<N;i++;&c[i][0]) {
    for(j=0;j<M;j++) {
      c[i][j]=0;
      for(l=0;l<P;l++) c[i][j] += a[i][l]*b[l][j]
    }
  }
}
```

MatLab with Distributed Toolbox

- MatLab with Distributed Computing Toolbox + MATLAB Distributed Computing Engine enable us to
  - develop distributed and parallel MatLab applications and
  - execute them on a cluster of computers without leaving your technical computing development
- The toolbox and engine based on the MPI
  - includes functions to support explicit communication, enabling you to develop parallel applications
  - supports parallel for loops and global array semantics via distributed arrays for creating parallel applications without explicit message passing

JEE with Web Services

- Java objects and service based approach
- Originally a distributed platform, can also be used for parallelism
- Purely explicit
- Easier to integrate with industrial projects

Workflows

- A dataflow-like approach applied to Grid environments
- Ready data (input, intermediate result) is transfered from its producer to consumer via file-transfer
  - kind of message-passing
- Modeled as a DAG where
  - each vertex represents a task, and
  - each edge represents a file-transfer operation

OpenMP + MPI + Workflows

- OpenMP → among the cores within a node
- MPI → among the nodes within a cluster
- Workflows → among the clusters within a Grid environment