Introduction

Ned Nedialkov

McMaster University
Canada

CS/SE 4F03
January 2016

Outline

von Neumann architecture

- Consists of
  - main memory
  - central processing unit (CPU)
  - interconnection between memory and CPU
- CPU
  - control unit
  - arithmetic and logic unit (ALU)
  - program counter (stores address of next instruction)
  - registers
- The interconnect determines the rate at which instructions and data are accessed
- Typically CPU executes much faster than accessing data
- SISD single instruction stream, single data stream

Processes

- Process: instance of a program
- Consists of
  - executable
  - stack
  - heap
  - file descriptors
  - information about which resources can be accesses
  - information about the state: ready, running, waiting, blocked
    see e.g.
    `http://en.wikipedia.org/wiki/Process_state`
    Try the `top` command
- Multitasking
  - many processes execute on a CPU (core)
  - the CPU switches between them

Threads

- smallest sequence of instructions that can be managed independently by a scheduler
- typically a thread is part of a process
- multiple threads can live within the same process
- share resources of a process, e.g. memory
- on a sinlge CPU, it switches between threads
- on multiple CPUs/cores, threads can execute in parallel

SIMD

SIMD, Single Instruction, Multiple Data

- the same instruction is applied to multiple data items
- example vector addition

```
for (i=0; i<n; i++)
        y[i] += x[i];
```

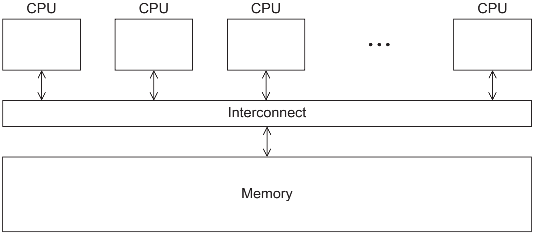
- vector processors: operate on arrays (or vectors)
  - popular in the 1970s through 90s, Cray, Thinking Machines
  - now SIMD instructions such as SSE, AltiVec
    Cell processor, GPUs

MIMD

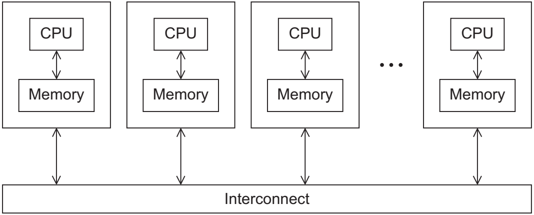
Multiple Instruction, Multiple Data

- independent CPU (cores)
- shared memory: each processor can access memory
- distributed memory
  - each processor has its own memory
  - communication is through message passing
- clusters or hybrid systems
  - nodes that are connected through an interconnection network
  - nodes are typically shared memory systems

Shared memory



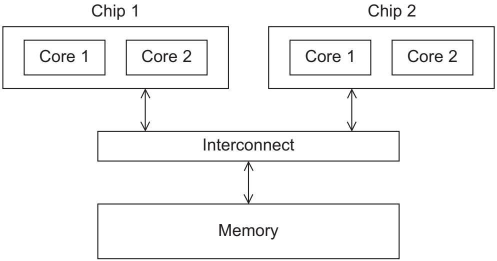
Distributed memory



Figures from P. Pacheco, An Introduction to Parallel Programming, 2011

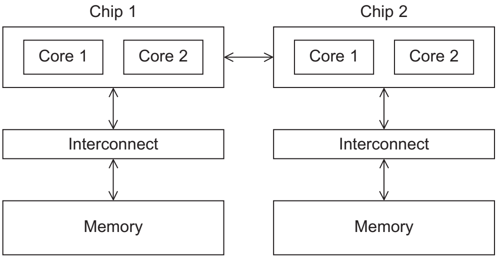
UMA vs. NUMA

- UMA: Uniform Memory Access
- Time to access main memory is the same for each CPU/core



- Most x86 multiprocessor systems, Xeon chips

- NUMA: Non-uniform memory access
- Memory local to a core is faster to access than non-local memory



- Example: https://www.sharcnet.ca/my/systems/show/15

SPMD

SPMD, Single Program, Multiple Data

- the same program executes as different processes
- a process knows its ID, usually called rank
- suppose a program P has

```
if (my_rank == 0) {
 A()
}
else {
 B()
}
```

If an instance of P executes as process 0, A() executes otherwise B() executes

Example: simple MPI program

```
/* Send a message from all processes with rank != 0 to
    process 0. Process 0 prints the messages received.
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
  int          my_rank;      /* rank of process      */
  int          p;            /* number of processes  */
  int          source;       /* rank of sender       */
  int          dest;         /* rank of receiver     */
  int          tag = 0;      /* tag for messages     */
  char         message[100]; /* storage for message  */
  MPI_Status   status;       /* status for receive   */
```

```
/* Initialize MPI */
MPI_Init(&argc, &argv);
/* Find out process rank  */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/* Find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &p);
if (my_rank != 0) {
  printf("PE␣%d␣sends␣␣␣␣to␣␣␣PE␣0:␣Hello␣from␣process
      ␣%d␣\n", my_rank, my_rank);
  /* Create message */
  sprintf(message,"Hello␣from␣process␣%d", my_rank);
  dest = 0;
  /* Use strlen+1 so that '\0' gets transmitted */
  MPI_Send(message, strlen(message)+1, MPI_CHAR,
           dest, tag, MPI_COMM_WORLD);
}
```

```
  else { /* my_rank == 0 */
    for (source = 1; source < p; source++)
      {
        MPI_Recv(message, 100, MPI_CHAR, source, tag,
                 MPI_COMM_WORLD, &status);
        printf("PE␣0␣receives␣from␣PE␣%d:␣%s\n", source,
             message);
      }
  }
  /* Shut down MPI */
  MPI_Finalize();
  return 0;
}
```

- Compile

```
mpicc mpi-greetings.c
```

- Execute on 4 processes

```
mpirun -np 4 ./a.out
```

Example: simple OpenMP program

```
#include <omp.h>
#include <stdio.h>
int main ()
{
int nthreads, tid;
/* Start threads */
#pragma omp parallel private(nthreads, tid)
  {
  /* Obtain thread number */
  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 */
}
```

- Compile

```
gcc -fopenm openmp-hello.c
./a.out
```

- In bash

```
export OMP_NUM_THREADS=8
./a.out
```

- In tcsh, csh

```
setenv OMP_NUM_THREADS 8
./a.out
```