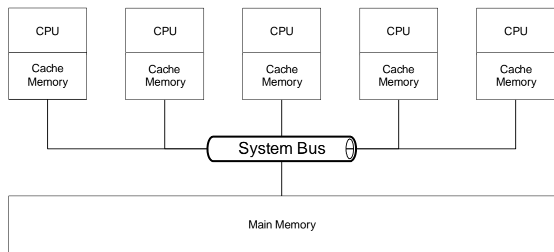
Left over

Review

- Introduction to parallel and distributed computing
- The need for parallel computation
- Parallel computer architectures

Shared-Memory Parallel Computer

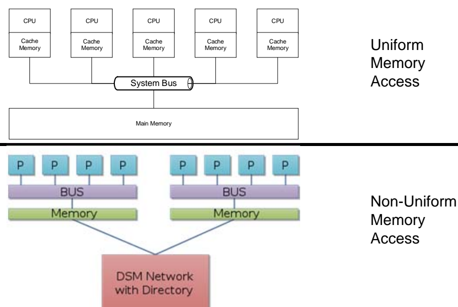


Shared-Memory Parallel Computer

- Shared Memory
  - Programming through threading
  - Multiple processors share a pool of memory
  - Problems: *cache coherence*
  - UMA vs. NUMA architecture
- Pros:
  - Easier to program (probably)
- Cons:
  - Performance may surfer if the memory is located on distant machines
  - Limited scalability

UMA vs. NUMA

http://en.wikipedia.org/wiki/Uniform_Memory_Access

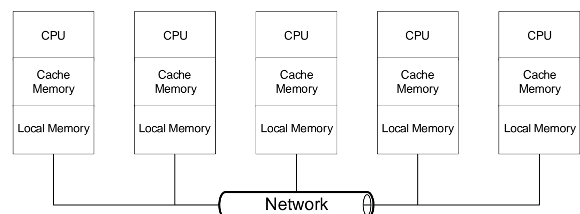


Uniform Memory Access

Non-Uniform Memory Access

http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access

Distributed-Memory Parallel Computer



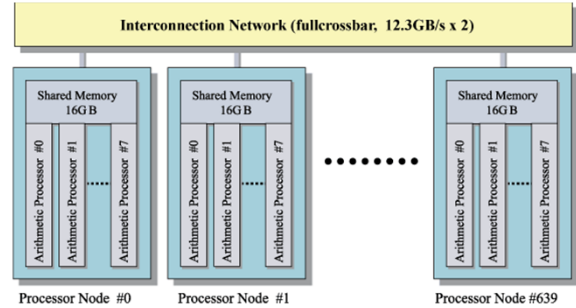
Distributed-Memory Parallel Computer

- Distributed Memory
  - Programming through processes
  - Explicit message passing
  - Networking
- Pros:
  - Tighter control on message passing
- Cons:
  - Harder to program
- **Modern supercomputers are hybrids!**

7

Earth Simulator

http://www.es.jamstec.go.jp/esc/eng/index.html
the fastest supercomputer in the world from 2002 to 2004



Beowulf Cluster

- Is a form of parallel computer
- Thomas Sterling, Donald Becker, … (1994)
- Emphasize the use of **COTS**
  - COTS: Components-Off-The-Shelf
  - Intel, AMD processors
  - Gigabit Ethernet (1000Mbps)
  - Linux
- A dedicated facility for parallel processing
  - Non-dedicated: NOW (Network Of Workstations)
- Performance/Price ratio is significantly higher than traditional supercomputers!

9

Models for Parallel Processing

10

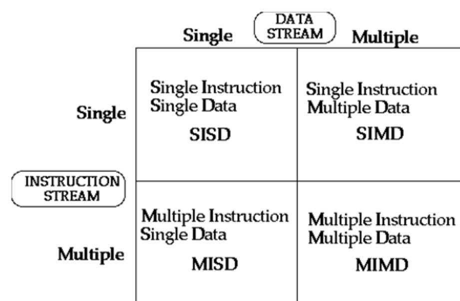
Models for Parallel Processing

- Flynn (1966)
  - SISD (Single Instruction, Single Data)
    - Conventional uni-processor machines
    - Serial, Deterministic
  - SIMD (Single Instruction, Multiple Data)
    - SSE, data parallel
    - Synchronous, Deterministic
    - ILP: Instruction Level Parallelism
  - MISD (Multiple Instruction, Single Data)
    - Rarely used
  - MIMD (Multiple Instruction, Multiple Data)
    - MPP (Massive Parallel Processing)
    - Synchronous or asynchronous
    - Deterministic or non-deterministic
    - Well-suited to block, loop, or subroutine level parallelism

5*(1, 2, 3, 4, 5) → (5, 10, 15, 20, 25)

11

Taxonomy of Computer Architectures (Flynn 1966)



12

Lecture 2

Misc Topics in
C/C++ Programming

Topics

- Dynamic memory allocation and memory layout
- Command Line Arguments
- Pre-processor Directives
- External Functions & Libraries
- Makefile

Dynamic Memory Allocation and Memory Layout

Dynamic Memory Allocation

- Array: A data structure consisting of a group of elements that are accessed by indexing. In most programming languages each element has the same data type and the array occupies a contiguous area of storage.
- Arrays are declared with a fixed size
  - `int a[5], b[10];`
  - `char c[]="12345";`
- Auto variables use memory space from stack, thus their sizes are limited!
- Dynamic variables use memory space from "heap", and their sizes are limited only by the available memory from OS (physical + virtual)
- Program often deals with unknown size of data set. Thus, we need to use dynamic memory management.

http://en.wikipedia.org/wiki/Array

Dynamic 1-D Array

```
int n;

cout << "Enter size: ";
cin >> n;

int *a = new int[n];
for(int i=0;i<n;i++) {
    a[i]=i;
}

delete []a;
```


/home/course/Lecture02/01.cpp

Dynamic 2-D Array

- How do we dynamically allocate memory space for multi-dimensional arrays?

```
#include <iostream>
using namespace std;
int main() {
        double a[][] = new double[20][10];

        for(int i=0;i<20;i++) {
                for(int j=0;j<10;j++) {
                        a[i][j]=0.0;
                }
        }
        return 0;
}
```

error: declaration of `a' as multidimensional array must have bounds for all dimensions except the first

/home/course/Lecture02/02.cpp

Look-alike 2-D Array

```
int m, n;

cout << "\nEnter m and n: ";
cin >> m >> n;

// allocation
double **a = new double* [m];
for(int i=0; i<m; i++) {
  a[i] = new double[n];
}
```

19

/home/course/Lecture02/03.cpp

Look-alike 2-D Array

```
// use
for(int i=0; i<m; i++) {
  for(int j=0; j<n; j++) {
    a[i][j] = i*n+j;
  }
}

// release
for(int i=0; i<m; i++) {
  delete []a[i];
}
delete []a;
```

20

/home/course/Lecture02/03.cpp

Pros and Cons of Look-alike 2-D Array

- Cons
  - Memory is NOT continuous.
  - Increases memory fragmentation.
  - Takes more effort (for OS) to allocate and release memory.
- Pros
  - Familiar looking
  - Less likely to make mistakes except for allocation and de-allocation

21

Simulated 2-D Array

```
int m, n;

cout << "\nEnter m and n (m rows and n columns): ";
cin >> m >> n;

// allocation
double *a = new double [m*n];
```

22

/home/course/Lecture02/04.cpp

Simulated 2-D Array

```
// use
for(int i=0; i<m; i++) {
  for(int j=0; j<n; j++) {
    a[i*n + j] = i*n + j;
  }
}

// release
delete []a;

return 0;
```

23

/home/course/Lecture02/04.cpp

Using Dynamic 2-D Arrays
Row-Major

- For 2-D arrays, we need to simulate it by managing array indices ourselves and store it in a one-dimensional array.

A[3][2] → A[3*nCol+2]

```
for(i=0;i<3;i++) {
  for(j=0;j<2;j++) {
    a[i][j]=0.0;
  }
}
```

```
for(i=0;i<3;i++) {
  for(j=0;j<2;j++) {
    a[i*nCol+j]=0.0;
  }
}
```

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[0]	A[1]	A[2]	A[3]

A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[4]	A[5]	A[6]	A[7]

A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[8]	A[9]	A[10]	A[11]

A[3][0]	A[3][1]	A[3][2]	A[3][3]
A[12]	A[13]	A[14]	A[15]

Using Dynamic 2-D Arrays
Column-Major

- For 2-D arrays, we need to simulate it by managing array indices ourselves and store it in a one-dimensional array.

A[3][2] → A[3+2*nRow]

```
for(j=0;j<2;j++) {
  for(i=0;i<3;i++) {
    a[i][j]=0.0;
  }
}
```

```
for(j=0;j<2;j++) {
  for(i=0;i<3;i++) {
    a[i+j*nRow]=0.0;
  }
}
```

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[0]	A[4]	A[8]	A[12]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[1]	A[5]	A[9]	A[13]
A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[2]	A[6]	A[10]	A[14]
A[3][0]	A[3][1]	A[3][2]	A[3][3]
A[3]	A[7]	A[11]	A[15]

Pros and Cons of Simulated 2-D Array

- Pros
  - Memory is contiguous.
  - Memory fragmentation is less likely.
  - Easier during memory allocation and de-allocation.
- Cons
  - Need to manually manage array indices.
  - Likely to make mistakes when managing indices

Cautions with Dynamic Memory Management

- Avoid frequent dynamic memory allocation!
  - Memory, like disk, can be fragmented.
  - It takes time for OS to find a contiguous memory block that meets the demand.
  - Memory fragments reduce maximum allocable memory.
- Be careful with memory leaks.
  - Allocating memory without releasing them.
  - As computation progresses, the available memory decreases, and it takes longer and longer to find available memory.
  - Not to mention memory swapping!
- *When you see "segmentation fault" or "sig11" when your program executes on Linux/UNIX, they are usually caused by programs using memory space that they are not supposed to use.*

Command Line Arguments

Command line arguments

- Purpose:
  - An alternative method of getting user-inputs
  - Taking inputs from CLI (command line interface)
    - cp **a.cpp** b.cpp ← a.cpp & b.cpp are arguments
  - Allowing the developed program use in conjunction with other programs, especially with shell scripting languages (bash, Perl, …)
  - Enables program running in batch without user intervention (through scripting)
  - This is especially important for using computer systems with queues.

Command Line Arguments

/home/course/Lecture02/05.cpp

```
#include <iostream>
using namespace std;

int main(int argc, char **argv) {
    for(int i=0;i<argc;i++) {
        cout << "\nargv[" << i << "]=" << argv[i];
    }
    return 0;
}
```

./a.out

argv[0]=./a.out

./a.out 1 2 3 4 5

argv[0]=./a.out
argv[1]=1
argv[2]=2
argv[3]=3
argv[4]=4
argv[5]=5

./a.out "Yo-Ming Hsieh" 1 49 B80501049

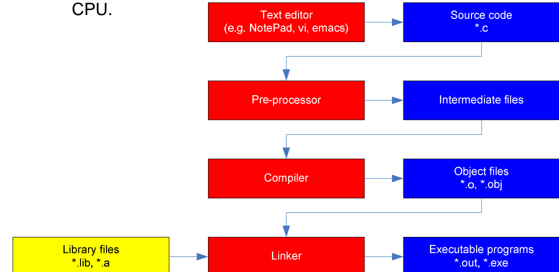
argv[0]=./a.out
argv[1]=Yo-Ming Hsieh
argv[2]=1
argv[3]=49
argv[4]=B80501049

Pre-processor Directives

#define / #undef macro
#include
conditional compilation

How executable files are generated

- Compiled
  - Programs written in C language are translated through compiler into binary format (machine language) that is understandable to CPU.



#define / #undef

- Define a macro with certain "value", and when source code is pre-processed, the macro/label will be replaced by its value.

/home/course/Lecture02/06.cpp

```
#define N 1000

int main() {
    int a[N][N];

    for(int i=0;i<N;i++) {
        for(int j=0;j<N;j++) {
            a[i][j]=0;
        }
    }
}

#undef N
```

```
int main() {
    int a[1000][1000];

    for(int i=0;i<1000;i++) {
        for(int j=0;j<1000;j++){
            a[i][j]=0;
        }
    }
}
```

#define / #undef

- Macro can also have arguments, and it looks like functions.

```
#define SQUARE(x) ((x) * (x))
#define MAX(x,y) ((x)>(y)?(x):(y))
#include <iostream>

int main() {
    std::cout << "\n5^2=" << SQUARE(5);
    std::cout << "\nMAX(3,5)=" << MAX(3,5);

    return 0;
}
```

/home/course/Lecture02/07.cpp

```
int main() {
    std::cout << "\n5^2=" << ((5) * (5)) ;
    std::cout << "\nMAX(3,5)=" << ((3)>(5)?(3):(5));

    return 0;
}
```


#include

- To add the content of other files into the current file at the place of the #include directive before compilation.

abc.inc

```
int n=3;
int j=4;
```

myProg.c

```
int main() {
#include "abc.inc"

    std::cout << "\nn=" << n << ", j=" << j;
    return 0;
}
```

```
int main() {
    int n=3;
    int j=4;

    std::cout << "\nn=" << n << ", j=" << j;
    return 0;
}
```


Conditional Compilation

- #ifdef … #else … #endif
- #ifndef … #else … #endif

/home/course/Lecture02/08.cpp

```
#define BETA
#include <iostream>
int main() {
#ifdef BETA
    std::cout << "\nThis is the beta version";
#else
    std::cout << "\nThis is the full version";
#endif
    return 0;
}
```

```
int main() {
    std::cout << "\nThis is the beta version";
    return 0;
}
```

Assignment #2

Due: 3/15/2011

37