Fundamentals of Parallel Processing

Dishant Ailawadi
IIT Roorkee
Tutor – Dr. Wellein and Mr. Gotz

8th Indo-German Winter Academy, Roorkee, December 2009
Outline

- Need for parallel computing
- Types of parallelism
- Parallel computer memory architectures
- Classes of parallel computers
- Key parallel processing steps & challenges
- CUDA and NVIDIA GPUs
- Cluster Computing & MPI
- Cloud computing
- Summary
Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem.

**Origin:** dates back to 1950s, in the mid 1980’s, a new kind of parallel computing was launched when the Caltech Concurrent Computation project built a supercomputer from 64 Intel 8086/8087 processors (an MPP)

In the late 80’s, clusters came to compete and eventually displace MPPs for many applications.

Today, parallel computing is becoming mainstream based on **multicore processors.** Increase performance by adding more cores.

For MPPs and clusters, a single standard called **MPI by the mid 1990’s.**

For shared memory systems: Pthreads & OpenMP by late 1990s

The key challenge in computing today is to transition the software industry to parallel programming.
Need for Parallel Computing

Moore’s Law:
Chip performance doubles every 18-24 months

Limits to serial computing
- Transmission speeds
- Limits to miniaturization
- Economic limitations
- Heating \[ P = C \times V^2 \times F \]
- Save time and/or money
- Solve larger problems: impossible on single computer (Web searching million transactions per second)
- Provide concurrency

Fundamentals of Parallel Processing
Modern Processors

- Pentium III
- Super-scalar: performs multiple operations on every clock cycle
- Out-of-order execution: doesn’t match ordering of assembly program
- Branch Prediction
  - `addl %eax,%edx` (1)
  - `addl %eax,4(%edx)` (3)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Issue Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Add</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>Floating Point Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Floating-Point Multiply</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Divide</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>Load (Cache Hit)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Store (Cache Hit)</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Types of parallelism

**Bit-level parallelism:**
- Increase in word size from 1970s to 1986.

**Instruction-level parallelism:**
Instruction pipelines
for(i=1 to n)
c=c*a[i] ;

**Data parallelism:**
Divide the dataset into grids or sectors and solve each sector on a separate execution unit.

**Task Parallelism:**
Divide the 'problem' into different tasks and execute the tasks on different units.
Flynn’s Classical Taxonomy

- **SISD**: A serial computer
  - load A
  - load B
  - C = A + B
  - store C
  - A = B * 2
  - store A

- **MISD**: Cryptographic Decoding
  - prev instruct
  - load A(1)
  - C(1) = A(1)*1
  - store C(1)
  - next instruct

- **SIMD**: GPUs
  - prev instruct
  - load A(1)
  - load B(1)
  - C(1) = A(1)*B(1)
  - store C(1)
  - next instruct

- **MIMD**: Clusters, Supercomputers
  - prev instruct
  - load A(1)
  - C(2) = A(1)*2
  - store C(2)
  - next instruct

- prev instruct
  - load A(n)
  - load B(n)
  - C(n) = A(n)*B(n)
  - store C(n)
  - next instruct

- prev instruct
  - load A(1)
  - C(1) = A(1)*B(1)
  - store C(1)
  - next instruct

- prev instruct
  - call funcD
  - x = y*z
  - sum = x*2
  - call sub1(i,j)
  - next instruct

- prev instruct
  - do 10 i=1,N
  - alpha = w**3
  - zeta = C(i)
  - 10 continue

- prev instruct
  - next instruct

---

Fundamentals of Parallel Processing 7
Concepts and Terminology

**Amdahl’s law:**

\[
\text{Speed Up} = \frac{1}{(1 - P) + \frac{P}{N}}
\]

**Dependencies:** No program can run more quickly than the longest chain of dependent calculations.

**Race conditions, mutual exclusion & synchronization**

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read X</td>
<td>Read X</td>
</tr>
<tr>
<td>X = X + 1</td>
<td>X = X + 1</td>
</tr>
<tr>
<td>Write X</td>
<td>Write X</td>
</tr>
</tbody>
</table>

**Granularity: Computation/Comm.**

- Fine-grained: Excessive Comm.
- Coarse-grained: Large granularity
- Embarrassing parallelism: Rare

Easily parallelizable

**Speed Up:**

- 1.67
- 2
Task A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

Parallel Task: Task executed by multiple processors safely (correct results)

Serial Execution: one statement at a time. what happens on a one processor

Parallel Execution: Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.

Communications: Exchanging data through a shared memory bus or over a network

Synchronization: Task may not proceed further until other reach same point.

Parallel Overhead: Task start-up/termination time; Synchronizations & data communications; Overhead imposed by parallel compilers, libraries, OS.

Scalability: Parallel system’s (S/W + H/W) ability to inc. speed up with inc. in no. of processors. Deciding factors: memory bandwidth; application algorithm; parallel overhead; characteristic of application.
Shared Memory Systems

Source Processor writes data to Global Mem & destination retrieves it.

- Easy to build, conventional OS of SISD can easily be ported
- Limitation: reliability & expandability. A memory component or any processor failure affects the whole system.
- Adding more processors increase traffic.
- Global address space proves user-friendly to programmers.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.
- Expensive: Difficult to design when more processors
- Programmer responsibility to insure "correct" access of global memory. (Race conditions)
Distributed Memory

- Network can be configured to Tree, Mesh, Cube.
- Unlike Shared MIMD
  - easily/ readily expandable
  - Highly reliable (any CPU failure does not affect the whole system)
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- The programmer is responsible for many of the details associated with data communication between processors.
Uniform memory access (UMA)

- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Non-Uniform Memory Access

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA
Classes of Parallel Computers

- **Multi-core computing**
  A multi-core processor is an integrated circuit to which two or more processors have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks.
  
  1. Cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods.
  2. Identical cores- *homogeneous*
     Not identical- *heterogeneous* multi-core systems.
Classes of Parallel Computers

- **Symmetric multiprocessing (SMP)**
  Hardware architecture where multiple processors share a single address space and access to all resources. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.

- Allow any processor to work on any task no matter where the data for that task are located in memory with proper operating system support,

- SMP systems can easily move tasks between processors to balance the workload efficiently.
Classes of parallel computers

- A **distributed** is a distributed memory computer system in which the processing elements are connected by a network.
- **Clusters** are composed of multiple standalone machines connected by a network.
- **MPP**: a single computer with many networked processors. Have specialized interconnect networks. Each CPU has its own memory & copy of OS. Eg. Blue Gene
- **Grid computing** is the most distributed form of parallel computing. It makes use of computers communicating over the Internet. deals only with embarrassingly parallel problems.
- **General-purpose computing on graphics processing units** is a fairly recent trend in computer engineering research. Act as co-processors.
Key Parallel Programming Steps

Parallel computing requires

To find the concurrency in the problem
1) To structure the algorithm so that concurrency can be exploited
2) To implement the algorithm in a suitable programming environment

Kruskal Algo: Select shortest edge of graph if it doesn’t make cycle. Repeat.
Sequence- (0,2) (3,4) (2,4) (1,2)

Prim’s Algo: Add chosen node to the set. Select edge with least weight originating from the set. Add edge to MST. Repeat.
Sequence- (0,2) (2,4) (3,4) (1,2)

Minimum Spanning Tree:
Baruvka’s Algorithm:
- Assign Each node and it’s edges to a processor
- Find shortest Edge from that node
- Combine connected components. Repeat

Fundamentals of Parallel Processing
An Example

- **PI Calculation**

  Serial pseudo code for this:

  ```plaintext
  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
  ```

  - Note that most of the time in running this program would be spent executing the loop
  - Leads to an embarrassingly parallel solution
    - Computationally intensive
    - Minimal communication
    - Minimal I/O
**Parallel strategy**: break the loop into portions that can be executed by the tasks.

- For the task of approximating PI:
  - Each task executes its portion of the loop a number of times.
  - Each task can do its work without requiring any information from the other tasks.
  - Uses the SPMD model. One task acts as master and collects the results.

```plaintext
npoints = 10000
circle_count = 0

p = number of processors  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
```
Challenges of Parallel Programming

- Looking at the problem from a non-obvious angle
- Dependences need to be identified and managed
  - The order of task execution may change the answers
- Performance can be drastically reduced by many factors
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Load imbalance among processor elements, non-synchronous threads.
  - Inefficient data sharing patterns
  - Saturation of critical resources such as memory bandwidth

- **Latency vs. Bandwidth**
  
  *Latency* is the time it takes to send a minimal message from point A to point B.
  
  *Bandwidth* is the amount of data that can be communicated per unit of time.

- It is more efficient to package small messages into a larger message.

- **Synchronous vs. asynchronous communications**
  
  Synchronous communications require some type of "handshaking". Blocking
  Asynchronous communications allow tasks to transfer data independently from one another. Non Blocking.
CUDA and NVIDIA GPUs

- CUDA is a software environment for parallel computing with minimal extensions to C/C++.
- Parallel portions of an application are executed on device as kernels, executing many threads.
- Enables heterogeneous systems (CPU+GPU)
- Shared, Global & Texture memory.
- CF= 600 MHz. but large no. of threads.

![CUDA and NVIDIA GPUs Diagram](image)
### Execution Model

#### Software
- Thread

#### Hardware
- Thread Processor
  - Threads are executed by thread processors
- Multiprocessor
  - Thread blocks are executed on multiprocessors
  - Thread blocks do not migrate
- Grid
  - Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
- Device
  - A kernel is launched as a grid of thread blocks
  - Only one kernel can execute on a device at one time
Data Movement Example

```c
int main(void) {
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i;
    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);
    for (i=0, i<N; i++) a_h[i] = 100.f + i;
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);
    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0; }
```
Increment Array Example

CPU program

```c
void inc_cpu(int *a, int N) {
    int idx;
    for (idx = 0; idx < N; idx++)
        a[idx] = a[idx] + 1;
}

int main() {
    ... inc_cpu(a, N);
}
```

CUDA program

```c
__global__ void inc_gupu(int *a, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) a[idx] = a[idx] + 1;
}

int main() {
    ... 
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize ) );
    inc_gpu<<<dimGrid, dimBlock>>>(a, N); }
```
Matrix Multiplication
Cluster Computing & MPI

- Group of linked computers working together closely.
- Cost Effective compared to other High Flops SC.
- Excellent for parallel operations but much poorer than traditional SCs for non parallel ops.
- MPI is used, Language independent comm. Protocol
- 1980s -early 1990s: Distributed memory, parallel computing architecture, but incompatible software tools for parallel programs need for a standard arose.
- Final version of draft released in May, 1994
- MPI-2 picked up where the first MPI specification left off, was finalized in 1996. Current MPI combine both.
- **Explicit parallelism:** Programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI
Message Passing Interface

- MPI conforms to the following rules:
  - Same program runs on all processes (SPMD) no restriction to MPMD
  - Synchronizes well with Data Parallelism.
  - No concept of shared memory
  - Enables parallel programs in C, Python (though seq. lang.)

Information about which processor is sending the message:
- Where is the data on the sending processor.
- What kind of data is being sent.
- How much data is there.
- Which processor(s) are receiving the message.
- Where should the data be left on the receiving processor.
- How much data is the receiving processor prepared to accept.
program mpitest
use MPI
integer rank, size, ierror

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
write(*,*) 'Hello World, I am ', rank, ' of ', size

call MPI_Finalize(ierr)
end
Automatic parallelization

- Converting sequential code into multi-threaded code in order to utilize multiple processors simultaneously.
- Most Focused: LOOPS
- An Example:  
  
  ```
  do i=1 to n
      z(i) = x(i) + y(i)
  enddo
  do i=2 to n
      z(i) = z(i-1)*2
  enddo
  ```

  ```
  do i=2 to n
      z(i) = z(1)*2^(i-1)
  enddo
  ```

  ```
  do i=2 to n
      z(i) = z(1)*2^(i-1)
  enddo
  ```

- Parallelizing compilers can be fairly successful with Fortran77 programs.
- Success with C has been more limited.
- Instead of parallelizing at compile time, it may be done at runtime as all the information about variable is available.
- If a compiler is trying to prove a certain relation in order to parallelize a given program loop, and is unable to prove it, the compiler could compile a runtime test into the program that can determine whether the relation is true.
Difficulties

- Dependence analysis is hard for code using indirect addressing, pointers, recursion, and indirect function calls.
- Can’t parallelize complex code, or global memory reads.
- Loops have an unknown number of iterations;
- Wrong results; degradation
- Less flexible than manual parallelization
- Different architectures
- Attempts:
  - Abstract Data and Communications Library (ADCL) Univ. of Stuttgart
  - Automatic performance tuning of MPI-parallel apps. (Runtime)
- Historic Compilers:
  - Vienna Fortran compiler
  - Polaris Compiler
Cloud Computing

- Instead of installing a suite of software for each computer, you'd only have to load one application.
- Application would allow workers to log into a Web-based service which hosts all the programs the user would need for his or her job.
- User applications like office, database applications etc. can all be put on the cloud.
- Google’s app engine allows one to upload his own applications on the net which everybody can use.
- Low Hardware Cost: Just need the cloud computing system's interface software,
- Communication costs and latency are too high to allow parallelization across the world. (Remote access of memory is costly)

- **distributed computing** **grid computing** **utility computing** **cloud computing**
Summary

- Types of parallelism
  - bit level
  - pipelines
  - data & functional

- Parallel computer memory architectures
  - Shared and distributed memory

- Key parallel processing steps & challenges

- Classes of parallel computers
  - MPPs, SMP, Clusters, Grid

- CUDA and NVIDIA GPUs
  - Data passing, Functional call

- Cluster Computing & MPI

- Cloud computing
References

- Lecs on parallel processing Dr. Wen-mei Hwu, UIUC
- **Computer Systems**, Randall bryant, CMU
- nVIDIA CUDA Programming guide ver 2.0
- **Concepts of High Performance Computing** Georg Hager, Gerhard Wellein
- Wikipedia articles
- Parallel Computing, Bhujade MR