Parallel Programming:
C thread, Open MP, and Open MPI

Presenter: Nasrin Sultana

Wichita State University
07/10/2012
Parallel Programming: Open MP, MPI, Open MPI & CUDA

Outline

- Parallel Programming
- Open MP
- MPI
- Open MPI
- CUDA
- Discussions
- Conclusions
Parallel Programming

**What is Parallel Programming**

The simultaneous use of more than one processor or computer to solve a problem by a program.

**Why Parallel Programming is necessary**

- Serial computing is too slow.
- Need for large amounts of memory not accessible by a single processor.

**Requirement for Parallel Programming**

Parallel Systems
Types of Parallel Computer

Shared Memory

P=Processor
M=Memory
Types of Parallel Computer

Distributed Memory

Distributed Memory

P=Processor
M=Memory
S=Switch
Types of Parallel Computer

Shared & Distributed Memory

Diagram:

- P = Processor
- M = Memory
- S = Switch

Node

Node
Terminology of Parallel Programming

Process

Process is basically *task*.

A *process* is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.
Terminology of Parallel Programming

**Thread**

A *thread of execution* is the smallest unit of processing that can be scheduled by an operating system.

On a *single processor*, multithreading generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads.

On a *multiprocessor* (including multi-core system), the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task.
C Thread
A C Thread object wraps around an actual thread of execution. It effectively defines how the task is to be executed—namely, at the same time as other threads.
Terminology of Parallel Programming

**Multicore Processor**

**Multi-core processor** is a single computing component with two or more independent actual processors (called "cores"), which are the units that read and execute program instructions.

The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing.

A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods.
**Terminology of Parallel Programming**

**Multicore Processor**

*Processors were originally developed with only one core.*

- A **dual-core processor** has two cores (e.g. AMD Phenom II X2, Intel Core Duo)
- A **quad-core processor** contains four cores (e.g. AMD Phenom II X4, Intel's quad-core processors, see i3, i5, and i7 at Intel Core),
- A **hexa-core processor** contains six cores (e.g. AMD Phenom II X6, Intel Core i7 Extreme Edition 980X),
- An **octa-core processor** contains eight cores (e.g. Intel Xeon E7-2820, AMD FX-8150)
**Terminology of Parallel Programming**

**Multithreading Processor**

*Simultaneous Multithreading*

Simultaneous multi-threading is the ability of a single physical processor to simultaneously dispatch instructions from more than one hardware thread context.

shows how *each cycle* an SMT processor selects instructions for execution from all threads. It exploits instruction-level parallelism by selecting instructions from any thread that can (potentially) issue. The processor then dynamically schedules machine resources among the instructions.
Multithreading Processor

**Coarse Grain Multithreading**

CGMT time shares hardware with explicit context switches between active threads. Only one thread at a time is allowed to dispatch instructions. To get better single thread performance, CGMT switches threads only when long latency events occur.

- Designate a “preferred” thread (e.g., thread A)
- Switch to thread B on thread A cache miss
- Switch back to A when A cache miss returns

**Fine Grain Multithreading**

FGMT time shares hardware by switching between threads every clock cycle. There is no explicit context switch since the thread contexts (register file and PC) are duplicated in hardware. Note that if there are not enough active threads, there will be empty time slots which degrades performance.

Switch threads every cycle (round-robin)
Challenges in Parallel Computing

Parallel Computing has both *hardware and software* challenges:

**Software Challenges:**
Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next is executed.
Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem.

**Hardware Challenges:**
- For parallel computing multicore processor is required.
- Load Balancing also another concern for parallel computing.
- GPU is required for graphics level parallel computing.
Parallel Programming Languages (PPL)

Platform of Parallel programming (C language)

Currently, the most commonly-used language for different systems. It is a high-level assembly language. It is very portable: compilers exist for virtually every processor. C language is the suitable platform for OpenMP and OpenMPI parallel programming.

OpenMP

Open Multi-Processing (OpenMP) is an application programming interface (API).
Parallel Programming Languages (PPL)

**MPI**

It’s not a programming language. Message Passing Interface (MPI) is a standardized and portable message-passing system. MPI programs always work with processes, but programmers commonly refer to the processes as processors.

**Open MPI**

The Open MPI is an open source MPI-2 implementation.

**CUDA**

CUDA is programming model/architecture and a parallel computing platform and invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).
Memory Supports by Different PPL

**OpenMP**
- multi-platform shared memory
- multiprocessing programming

**MPI & Open MPI**
- Both programs are regularly run on distributed and shared memory computers

**CUDA**
- It can perform on both distributed and shared memory computers.
Platforms of PPL

- C programming language
- Fortran
- Java
- Python
Matrix Multiplication

How Matrix Multiplication works

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{pmatrix} \times \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \\ b_{2,0} & b_{2,1} \end{pmatrix}$$

Result

$$c_{0,0} = a_{0,0} b_{0,0} + a_{0,1} b_{1,0} \quad c_{0,1} = a_{0,0} b_{0,1} + a_{0,1} b_{1,1} \quad c_{0,2} = a_{0,0} b_{0,2} + a_{0,1} b_{1,2}$$
$$c_{1,0} = a_{1,0} b_{0,0} + a_{1,1} b_{1,0} \quad c_{1,1} = a_{1,0} b_{0,1} + a_{1,1} b_{1,1} \quad c_{1,2} = a_{1,0} b_{0,2} + a_{1,1} b_{1,2}$$
$$c_{2,0} = a_{2,0} b_{0,0} + a_{2,1} b_{1,0} \quad c_{2,1} = a_{2,0} b_{0,1} + a_{2,1} b_{1,1} \quad c_{2,2} = a_{2,0} b_{0,2} + a_{2,1} b_{1,2}$$
$$c_{3,0} = a_{3,0} b_{0,0} + a_{3,1} b_{1,0} \quad c_{3,1} = a_{3,0} b_{0,1} + a_{3,1} b_{1,1} \quad c_{3,2} = a_{3,0} b_{0,2} + a_{3,1} b_{1,2}$$
OpenMP

Motivations

- Allow a programmer to separate a program into serial regions and parallel regions.
- OpenMP can enable (easy) parallelization of loop-based code
- Hide stack management
- Provide synchronization constructs
- Helps to run short program within short time.
- Proceed long program within short span of time.
A master thread spawns teams of threads as needed.

Parallelism is added incrementally; the serial program evolves into a parallel program.
Serial Program

- With C program
- With C threads

Procedure is called by main()
Two (relatively) simple steps:

- Initialize matrix
- Call function
```c
if( clock_gettime( CLOCK_REALTIME, &start) == -1 ) {
    perror( "clock_gettime" );
    exit( EXIT_FAILURE );
}

matrixMul();

if( clock_gettime( CLOCK_REALTIME, &stop) == -1 ) {
    perror( "clock_gettime" );
    exit( EXIT_FAILURE );
}

void matrixMul()
{
    int i, j;
    int row, col;
    double mysum, *x, *y;

    for (i=0; i<matstr.len ; i++)
    {
        row = (i==0? 0: i/matstr.x);
        col = (i==0? 0: i%matstr.y);
        for (j=0; j<matstr.x; j++){
            matstr.c[col*matstr.x + row] += (matstr.b[col*matstr.x + j] * matstr.a[matstr.y*j + row]);
        }
    }
```
C program with Threads

Generates threads in C programs, does it in 3 steps

- Initialize and Divide data for each thread
- Calls function for each thread
- Compute
/* 3. Compute handling size of each thread */
mulstr_len = xy / NUMTHRDS;
for(i=0; i<NUMTHRDS; i++)
{
    /* Each thread works on a different set of data. 
    * Pass i to matrixAdd, telling which thread it is 
    */
    pthread_create(&callThd[i], &attr, matrixMul, (void *)i);
}

void *matrixMul(void *arg)
{
    int i, j, start, end, len;
    int row, col;
    long offset;
    double mysum, *x, *y;
    offset = (long)arg;

    len = mulstr_len;
    start = offset*len;
    end = start + len;

    for (i=start; i<end; i++)
    {
        row = (i==0? 0: i/mulstr.x);
        col = (i==0? 0: i%mulstr.y);
        for (j=0; j<mulstr.x; j++)
        {
            mulstr.c[col*mulstr.x + row] +=
        }
    }
}
Program with OpenMP

OpenMP shortens thread creation, reduces 3 steps to create threads

- Initialize (like in C)
- Compute (OpenMP divides tasks for each thread)
OpenMP Program

```c
#pragma omp parallel shared(nthreads) private(tid,i,j,k)
{
    if (tid == 0)
    {
        nthreads = omp_get_num_threads(); //for display purposes
    }
#pragma omp for schedule(static)
    for (i=0; i<mulstr.x ; i++) {
        for (j=0; j<mulstr.y; j++)
            for (k=0; k<mulstr.y; k++)
                mulstr.c[i*mulstr.x + j] += (mulstr.b[i*mulstr.x + k] * mu
```
Performance Evaluation of C, Cthread and OpenMP

The diagram shows the execution time (in seconds) for different configurations of cores and threads. The x-axis represents the number of threads, while the y-axis represents the execution time. The legend indicates the number of cores and whether threads are used. For example:

- 1 core with C threads
- 2 cores with C threads
- 4 cores with C threads
- 8 cores with C threads
- 1 core with OpenMP
- 2 cores with OpenMP
- 4 cores with OpenMP
- 8 cores with OpenMP
- 2 cores without threads
- 4 cores without threads
- 8 cores without threads

The graph illustrates how the execution time changes with different numbers of threads and cores, highlighting the performance impact of using C, Cthread, and OpenMP.
Motivations

- High performance: It distributes the tasks among the processors/cores.
- Scalability: It can handle any size of problem.
- Portability: It is independent of language.
- It uses distributed memory so that the execution is faster of a program.
- Emphasizes message passing and has a static runtime environment.
- MPI-2 includes new features such as parallel I/O, dynamic process management and remote memory operations.
MPI uses message passing technique to send and receive data. Message passing involves the transfer of data from one process (send) to another process (receive). Requires the cooperation of both the sending and receiving process. Send operations usually require the sending process to specify the data's location, size, type and the destination. Receive operations should match a corresponding send operation.
An envelope portion

- The exact definition depends on the implementation
- Typically consists of the message tag, communicator, source, destination, and possibly the message length

A data portion

- Contains the information to be passed
- The exact definition depends on the implementation
- Using standard or derived data types
MPI Model

MPI include file

Declarations, prototypes, etc.

Program Begins

. Serial code

. Initialize MPI environment

Parallel code begins

. .

Do work & make message passing calls

. .

Terminate MPI environment

Parallel code ends

. Serial code

Program Ends
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.
Example of Point-to-Point Communication

1. This example performs communication between two processors.
2. Processor 0 sends a message Hello, World using blocking MPI_Send to processor 1, which receives this message using blocking receive MPI_Recv.
3. NO MATTER HOW MANY PROCESORS ARE THERE.
Collective communication must involve all processes in the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD.

It is the programmer's responsibility to insure that all processes within a communicator participate in any collective operations.
Example of Collective Communication

```c
void my_bcast(void* data, int count, MPI_Datatype datatype, MPI_Comm communicator) {
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);

    if (world_rank == root) {
        // If we are the root process, send our data to everyone
        int i;
        for (i = 0; i < world_size; i++) {
            if (i != world_rank) {
                MPI_Send(data, count, datatype, i, 0, communicator);
            }
        }
    } else {
        // If we are a receiver process, receive the data from the root
        MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
    }
}
```

```bash
> make
mpiC -o my_bcast my_bcast.c
./run perl my_bcast
mpirun -n 4 ./my_bcast
Process 0 broadcasting data 100
Process 2 received data 100 from root process
Process 3 received data 100 from root process
Process 1 received data 100 from root process
```
Comparison of OpenMP, Pthread and OpenMPI

In OpenMP and pThread
- all threads work on the same data
- no duplication

OpenMPI
- Thread requires communications
- Relatively complicated
- Let's see the steps
Step 1

User calls MPI program using mpiexec
-mpiexec -np (4) <program name>
OS spawns 4 processes on selected nodes
The Code-step1

```c
/* Send A by splitting it in row-wise parts */
if (!rank)
{
    alreadysent = upperBound;
    for (i=1; i<sizem; i++)
    {
        tosent = nl * getRowCount(n, i, sizem);
        MPI_Send(A + alreadysent, tosent, MPI_DOUBLE, i, TAG_INIT, MPI_COMM_WORLD);
        alreadysent += tosent;
    }
}
else { /* Receive parts of A */
    MPI_Recv(A, upperBound, MPI_DOUBLE, 0, TAG_INIT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Get_processor_name(processor_name, &namelen);
    //printf("Processor %d sending from %s\n", rank, processor_name);
}
```

If rank 0, send matrices data to other processes
If rank not 0, send receive matrices data
Note that code above are executed in each process everytime it is spawned
Step 2

Each process compute previously divided data
The code-Step 2

```c
/* Let each process perform its own multiplications */
matrixMultiply(A, B, C, n, local,n1,n2);

int matrixMultiply(int *a, int *b, int *c, int n, int local,int n1,int n2) {
    int i, j, k;
    for (i=0; i<local; i++) {
        for (j=0; j<n2; j++) {
            for (k=0; k<n1; k++) {
                c[i*n + j] += a[i*n + k] * b[k*n1 + j];
            }
        }
    }
    return 0;
}
```

Each process calls the function
Note that received data are already divided before sending to minimize communication overhead
The code-Step3

/* Receive partial results from each slave */
if (!rank) {
    alreadysent = upperBoundc;
    for (i=1; i<sizem; i++) {
        tosent = n2 * getRowCount(n, i, sizem);
        MPI_Recv(C + alreadysent, tosent, MPI_DOUBLE, i, TAG_RESULT,
                  alreadysent += tosent;
    }
}
else { /* Send partial results to master */
    MPI_Send(C, upperBoundc, MPI_DOUBLE, 0, TAG_RESULT, MPI_COMM_WORLD);
}

/* Stop timer, includes communication time */
if(!rank){
    t = MPI_Wtime() - t;
}
if(!rank){
    printf("Total time for processor %d was %f seconds.\n", rank, t2);
    printf("Total time including communication was %f seconds.\n", t);
}
/* Goodbye, world */
MPI_Finalize();
return 0;

Rank 0:
MPI_Recv writes received data to matrix C (C is an address)
Rank 1:
Send data to rank 0
Discussions

Limitations of Open MP

- OpenMP does not parallelize dependencies
- OpenMP is not guaranteed to divide work optimally among threads

Advantage of Open MP

- Supported by a large number of compilers
- Requires little programming effort
- Allows the program to be parallelized incrementally
Discussion

Limitations of MPI

- It does not provide support for active message and fault tolerance.

Advantage of Open MP

- As Allow efficient communication in a heterogeneous environment
- Assume a reliable and stable communication interface
- Enables standard interface implementation by multiple platforms.
- Provides language independent interface
- The interface should be designed to allow for thread-safety
Discussion

Advantage of Open MPI
- Open MPI is established for the promotion, dissemination, and use of open source software in high-performance computing.

Advantage of CUDA
- Huge increase in processing power over conventional CPU processing. Early reports suggest speed increases of 10x to 200x over CPU processing speed.
- All graphics cards in the G80 series and beyond support CUDA.
- Harnesses the power of the GPU by using parallel processing; running thousands of simultaneous reads instead of single, dual, or quad reads on the CPU.
In a shared-memory architecture, there is usually one process which contains couple of threads which share the same memory address space, file handles and so on. Hence, the shared memory name. In this architecture, each threads can modify a "precess" global data. Therefore, a semaphore mechanism must be in use. OpenMP simplify the programming for shared memory architerture by providing compiler "extensions" in the form of various standardized "pragma"s.
Conclusions

➢ In a distributed-memory architecture, each process doesn't share the same address space as the other process (which very possibly run on different machine). This means each process cannot "see" the other process variable(s). The process must "send a message" to the other process to change variable in the other process. Hence, the "Massage Passing Interface (MPI)". The MPI library such as OpenMPI basically is a sort of "middleware" to facilitate the massage passing between the processes, the process migration, initialization and tear-down.
Conclusions

- CUDA provides ability to use high-level languages such as C to develop application that can take advantage of high level of performance and scalability that GPUs architecture offer. CUDA also exposes fast shared memory (16KB) that can be shared between threads. Full support for integer and bitwise operations.
Parallel Programming:
C thread, Open MP, and Open MPI

THANK YOU