Parallel Paradigms & Programming Models

Lectured by: Pham Tran Vu

Prepared by: Thoai Nam

Outline

- Parallel programming paradigms
- Programmability issues
- Parallel programming models
  - Implicit parallelism
  - Explicit parallel models
  - Other programming models

Parallel Programming Paradigms

- Parallel programming paradigms/models are the ways to
  - Design a parallel program
  - Structure the algorithm of a parallel program
  - Deploy/run the program on a parallel computer system
- Commonly-used algorithmic paradigms
  - Phase parallel
  - Synchronous and asynchronous iteration
  - Divide and conquer
  - Pipeline
  - Process farm
  - Work pool

Parallel Programmability Issues

- The programmability of a parallel programming models is
  - How much easy to use this system for developing and deploying parallel programs
  - How much the system supports for various parallel algorithmic paradigms
- Programmability is the combination of
  - Structuredness
  - Generality
  - Portability

Structuredness

- A program is *structured* if it is comprised of *structured constructs* each of which has these 3 properties
  - Is a single-entry, single-exit construct
  - Different semantic entities are clearly identified
  - Related operations are enclosed in one construct
- The structuredness mostly depends on
  - The programming language
  - The design of the program

Generality

- A program class C is as general as or more general than program class D if:
  - For any program Q in D, we can write a program P in C
  - Both P & Q have the same semantics
  - P performs as well as or better than Q

Portability

- A program is portable across a set of computer system if it can be transferred from one machine to another with little effort
- Portability largely depends on
  - The language of the program
  - The target machine's architecture
- Levels of portability
  1. Users must change the program's algorithm
  2. Only have to change the source code
  3. Only have to recompile and relink the program
  4. Can use the executable directly

Parallel Programming Models

- Widely-accepted programming models are
  - Implicit parallelism
  - Data-parallel model
  - Message-passing model
  - Shared-variable model (Shared Address Space model)

Implicit Parallelism

- The compiler and the run-time support system automatically exploit the parallelism from the sequential-like program written by users
- Ways to implement implicit parallelism
  - Parallelizing Compilers
  - User directions
  - Run-time parallelization

Parallelizing Compiler

- A parallelizing (restructuring) compiler must
  - Performs dependence analysis on a sequential program's source code
  - Uses transformation techniques to convert sequential code into native parallel code
- Dependence analysis is the identifying of
  - Data dependence
  - Control dependence

Parallelizing Compiler(cont'd)

- Data dependence

```
X = X + 1
Y = X + Y
```

- Control dependence

```
If f(X) = 1 then Y = Y + Z;
```

- When dependencies do exist, transformation techniques/ optimizing techniques should be used
  - To eliminate those dependencies or
  - To make the code parallelizable, if possible

Some Optimizing Techniques for Eliminating Data Dependencies

- Privatization technique

```
Do i=1,N
P:    A  = …
Q:    X(i)= A + …
      …
End Do
```

```
ParDo i=1,N
P:    A(i)= …
Q:    X(i) = A(i) + …
      …
End Do
```

Q needs the value **A** of P, so N iterations of the Do loop can not be parallelized

Each iteration of the Do loop have a private copy **A(i)**, so we can execute the Do loop in parallel

Some Optimizing Techniques for Eliminating Data Dependencies(cont'd)

- Reduction technique

```
Do i=1,N
P:      X(i)  =  …
Q:      Sum = Sum + X(i)
        …
End Do
```

The Do loop can not be executed in parallel since the computing of Sum in the i-th iteration needs the values of the previous iteration

```
ParDo i=1,N
P:      X(i)  = …
Q:      Sum = sum_reduce(X(i))
        …
End Do
```

A parallel reduction function is used to avoid data dependency

User Direction

- Users help the compiler in parallelizing by
  - Providing additional information to guide the parallelization process
  - Inserting compiler directives (pragmas) in the source code
- User is responsible for ensuring that the code is correct after parallelization
- Example (Convex Exemplar C)

```
#pragma_CNX loop_parallel
for (i=0; i <1000;i++){
        A[i] = foo (B[i], C[i]);
}
```

Run-Time Parallelization

- Parallelization involves both the compiler and the run-time system
  - Additional construct is used to decompose the sequential program into multiple tasks and to specify how each task will access data
  - The compiler and the run-time system recognize and exploit parallelism at both the compile time and run-time
- Example: Jade language (Stanford Univ.)
  - More parallelism can be recognized
  - Automatically exploit the irregular and dynamic parallelism

Conclusion - Implicit Parallelism

- Advantages of the implicit programming model
  - Ease of use for users (programmers)
  - Reusability of old-code and legacy sequential applications
  - Faster application development time
- Disadvantages
  - The implementation of the underlying run-time systems and parallelizing compilers is so complicated and requires a lot of research and studies
  - Research outcome shows that automatic parallelization is not so efficient (from 4% to 38% of parallel code written by experienced programmers)

Explicit Programming Models

- Data-Parallel
- Message-Passing
- Shared-Variable

Data-Parallel Model

- Applies to either SIMD or SPMD modes
- 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

Data-Parallel: An Example

Example: a data-parallel program to compute the constant “pi”

```
main() {
    double local[N], tmp[N], pi, w;
    long i, j, t, N=100000;
    A: w=1.0/N;
    B: forall(i=0; i<N; i++) {
        P: local[i]=(i +0.5)*w;
        Q: tmp[i]=4.0/(1.0+local[i]*local[i]);
       }
    C: pi=sum(tmp);
    D: printf(“pi is %f\n”, pi*w);
} //end main
```

Data-parallel operations

Reduction operation

Message-Passing Model

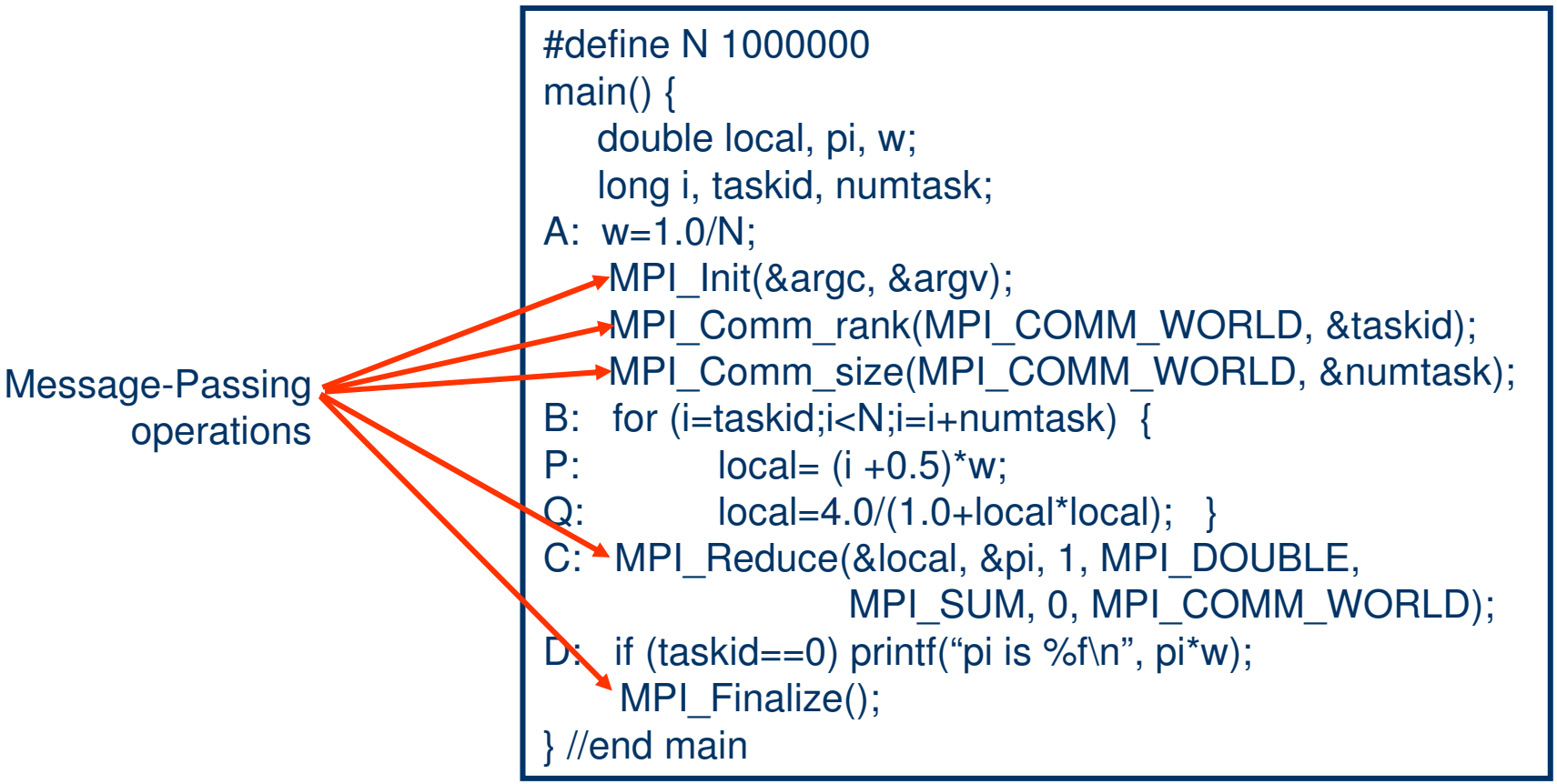
- Multithreading: program consists of multiple processes
  - Each process has its own thread of control
  - Both control parallelism (MPMD) and data parallelism (SPMD) are supported
- Asynchronous Parallelism
  - All process execute asynchronously
  - Must use special operation to synchronize processes
- Multiple Address Spaces
  - Data variables in one process is invisible to the others
  - Processes interact by sending/receiving messages

Message-Passing Model (cont'd)

- 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

Message-Passing Model: An Example

Example: a message-passing program to compute the constant "pi"



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

Shared-Variable Model: An Example

```
#define N 1000000
main() {
      double local, pi=0.0, w;
      long i;
A:    w=1.0/N;
B:    #pragma parallel
      #pragma shared (pi,w)
      #pragma local(i,local)
      {
            #pragma pfor iterate (i=0;N;1)
            for(i=0;i<N;i++){
P:                local= (i +0.5)*w;
Q:                local=4.0/(1.0+local*local);
            }
C:          #pragma critical
                  pi=pi+local;
       }
D:    if (taskid==0) printf("pi is %f\n", pi*w);
} //end main
```

Comparision of Four Models

Issues	Implicit	Data-parallel	Message-passing	Shared-Variable
Platform-independent examples	Kap, Forge	Fortran 90, HPF, HPC++	PVM, MPI	X3H5
Platform-dependent examples		CM C*	SP2 MPL, Paragon Nx	Cray Craft, SGI Power C
Parallelism issues	★ ★ ★ ★	★ ★ ★	★	★ ★
Allocation issues	★ ★ ★ ★	★ ★	★	★ ★ ★
Communication	★ ★ ★ ★	★ ★ ★	★	★ ★ ★
Synchronization	★ ★ ★ ★	★ ★ ★ ★	★ ★	★
Aggregation	★ ★ ★ ★	★ ★ ★	★ ★ ★ ★	★
Irregularity	★ ★ ★ ★	★ ★	★ ★	★ ★ ★
Termination	★ ★ ★ ★	★ ★ ★ ★	★ ★	★
Determinacy	★ ★ ★ ★	★ ★ ★ ★	★ ★	★
Correctness	★ ★ ★ ★	★ ★ ★	★ ★	★
Generality	★	★ ★	★ ★ ★	★ ★ ★
Portability	★ ★ ★ ★	★ ★ ★	★ ★	★
Structuredness	★ ★ ★ ★	★ ★	★	★

Comparision of Four Models (cont'd)

- 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

Comparision of Four Models (cont'd)

- 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
  - Be widely-accepted
  - Exploit large-grain parallelism and can be executed on machines with native shared-variable model (multiprocessors: DSMs, PVPs, SMPs)
- Shared-variable model
  - No widely-accepted standard → programs have low portability
  - Programs are more difficult to debug than message-passing programs

Other Programming Models

- Functional programming
- Logic programming
- Computing-by-learning
- Object-oriented programming