Principles of Parallel Algorithm Design

1

Why is Parallel Computing Hard?

- Amdahl's law – insufficient available parallelism –
  - Speedup = 1/(fraction_enhanced/speedup_enhanced + (1-fraction_enhanced))
- Overhead of communication and coordination
- Portability – knowledge of underlying architecture often required

2

Parallel Programming Models

Parallel program: one or more threads of control operating on data; model defines naming, operations, ordering

- Data parallel – HPF, Fortran-D, Power C/Fortran
- Shared memory - pthreads
- Message passing – MPI, PVM
- Global address space

3

- Task: arbitrarily defined piece of work done by the program
- Process (or thread): abstract entity that performs tasks
- Processor: phyical resource on which processes execute

4

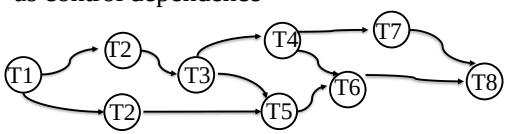
Steps in the Parallelization

- Decomposition into tasks
  - Expose concurrency
- Assignment to processes
  - Balancing load and maximizing locality
- Orchestration
  - Name and access data
  - Communicate (exchange) data
  - synchronization among processes
- Mapping
  - Assignment of processes to processors

5

Decomposition into Tasks

- Many different decompositions possible
  - Tasks may be independent or have dependencies requiring ordering
  - Tasks may execute identical or different code
  - Tasks may take the same or different amounts of time
- Tasks and dependencies may be abstracted into a task dependency DAG with nodes as tasks, edges as control dependence

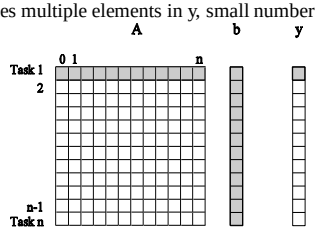


6

Granularity of Task Decompositions

- Task size (granularity) versus number of tasks

Example: Dense matrix-vector multiply
Fine grain: each task computes an individual element in y, large number of tasks
Coarse grain: each task computes multiple elements in y, small number of tasks



7

Degree of Concurrency

- *Degree of concurrency* of a decomposition: number of tasks that can execute in parallel
  - Increases with finer task granularity
- May change over program execution
- *Maximum degree of concurrency:* the largest at any point during execution
- *Average degree of concurrency:* average over the execution of the program

8

Critical Path

- Edge in task dependency graph represents task serialization
- Critical path = longest weighted path though graph
- Critical path length = lower bound on parallel execution time

9

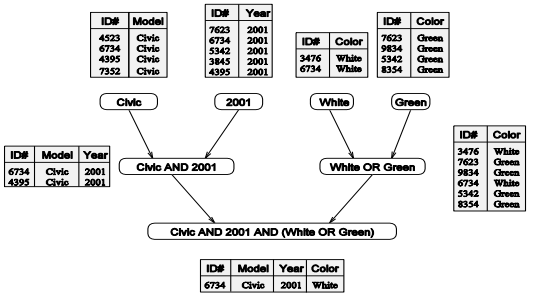
Example: Database Query Processing

Query:

MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

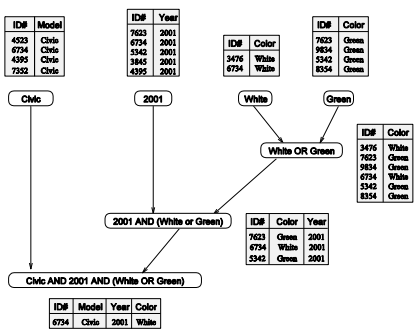
Task: generate intermediate table that satisfy predicate

Edge: output from task at outgoing edge feeds input to task at incoming edge



10

Database Query Processing: Alternative decomposition



Different decompositions may lead to different parallel performance

11

Basics of Parallelization

- Dependence analysis
- Synchronization
  - Events
  - Mutual exclusion
- Parallelism patterns

12

When can two statements execute in parallel?

- On one processor:

```
statement 1;
statement 2;
```

- On two processors:

```
processor1:          processor2:
    statement1;          statement2;
```

13

Fundamental Assumption

- Processors execute independently: no control over order of execution between processors

14

When can 2 statements execute in parallel?

- Possibility 1

```
Processor1:          Processor2:
    statement1;
                         statement2;
```

- Possibility 2

```
Processor1:          Processor2:
                         statement2:
    statement1;
```

15

When can 2 statements execute in parallel?

- Their order of execution must not matter!
- In other words,

```
statement1; statement2;
```

must be equivalent to

```
statement2; statement1;
```

16

Example 1

```
a = 1;
b = 2;
```

- Statements can be executed in parallel.

17

Example 2

```
a = 1;
b = a;
```

- Statements cannot be executed in parallel
- Program modifications may make it possible.

18

Example 3

```
a = f(x);
b = a;
```

- May not be wise to change the program (sequential execution would take longer).

19

Example 4

```
b = a;
a = 1;
```

- Statements cannot be executed in parallel.

20

Example 5

```
a = 1;
a = 2;
```

- Statements cannot be executed in parallel.

21

True dependence

Statements S1, S2

S2 has a true dependence on S1
iff
S2 reads a value written by S1

22

Anti-dependence

Statements S1, S2.

S2 has an anti-dependence on S1
iff
S2 writes a value read by S1.

23

Output Dependence

Statements S1, S2.

S2 has an output dependence on S1
iff
S2 writes a variable written by S1.

24

Types of Dependences

- True (flow) dependence – RAW
- Anti-dependence – WAR
- Output dependence – WAW

25

When can 2 statements execute in parallel?

S1 and S2 can execute in parallel
iff
there are no dependences between S1 and S2

- true dependences
- anti-dependences
- output dependences

Some dependences can be removed.

26

Example 6

- Parallelism often occurs in loops.

```
for(i=0; i<100; i++)
    a[i] = i;
```

- No dependences.
- Iterations can be executed in parallel.

27

Example 7

```
for(i=0; i<100; i++) {
   a[i] = i;
   b[i] = 2*i;
}
```

Iterations and statements can be executed in parallel.

28

Example 8

```
for(i=0;i<100;i++) a[i] = i;
for(i=0;i<100;i++) b[i] = 2*i;
```

Iterations and loops can be executed in parallel.

29

Example 9

```
for(i=0; i<100; i++)
    a[i] = a[i] + 100;
```

- There is a dependence … on itself!
- Loop is still parallelizable.

30

Example 10

```
for( i=0; i<100; i++ )
  a[i] = f(a[i-1]);
```

- Dependence between a[i] and a[i-1].
- Loop iterations are not parallelizable.

31

Loop-Carried Dependence

- A loop-carried dependence is a dependence that is present only if the statements occur in two different instances of a loop
- Otherwise, we call it a loop-independent dependence
- Loop-carried dependences limit loop iteration parallelization

32

Example 11

```
for(i=0; i<100; i++ )
  for(j=0; j<100; j++ )
      a[i][j] = f(a[i][j-1]);
```

- Loop-independent dependence on i.
- Loop-carried dependence on j.
- Outer loop can be parallelized, inner loop cannot.

33

Example 12

```
for( j=0; j<100; j++ )
  for( i=0; i<100; i++ )
      a[i][j] = f(a[i][j-1]);
```

- Inner loop can be parallelized, outer loop cannot.
- Less desirable situation (finer-grain parallelism).
- Loop interchange is sometimes possible.

34

Level of loop-carried dependence

- Is the nesting depth of the loop that carries the dependence.
- Indicates which loops can be parallelized.

35

Be careful … Example 13

```
printf("a");
printf("b");
```

Statements have a hidden output dependence due to the output stream.

36

Be careful … Example 14

```
a = f(x);
b = g(x);
```

Statements could have a hidden dependence if f and g update the same variable.

37

Be careful … Example 15

```
for(i=0; i<100; i++)
  a[i+10] = f(a[i]);
```

- Dependence between a[10], a[20], …
- Dependence between a[11], a[21], …
- …
- Some parallel execution is possible.

38

Be careful … Example 16

```
for( i=1; i<100;i++ ) {
        a[i] = …;
        ... = a[i-1];
}
```

- Dependence between a[i] and a[i-1]
- Complete parallel execution impossible
- Pipelined parallel execution possible

39

Be careful … Example 17

```
for( i=0; i<100; i++ )
        a[i] = f(a[indexa[i]]);
```

- Cannot tell for sure.
- Parallelization depends on user knowledge of values in indexa[].
- User can tell, compiler cannot.

40

An aside

- Parallelizing compilers analyze program dependences to decide parallelization.
- In parallelization by hand, user does the same analysis.
- Compiler more convenient and more correct
- User more powerful, can use knowledge of data values

41

To remember

- Statement order must not matter.
- Statements must not have dependences.
- Some dependences can be removed.
- Some dependences may not be obvious.

42

Synchronization

- Used to enforce dependences
- Control the ordering of events on different processors
  - Events – signal(x) and wait(x)
  - Fork-Join or barrier synchronization (global)
  - Mutual exclusion/critical sections

43

Example 1: Creating Parallelism by Enforcing Dependences

```
for( i=1; i<100; i++ ) {
    a[i] = …;
    …;
    … = a[i-1];
}
```

- Loop-carried dependence, not parallelizable

44

Synchronization Facility

- Suppose we had a set of primitives, signal(x) and wait(x).
- wait(x) blocks unless a signal(x) has occurred.
- signal(x) does not block, but causes a wait(x) to unblock, or causes a future wait(x) not to block.

45

Example 1: Enforcing Dependencies (continued)

```
for( i=...; i<...; i++ ) {
    a[i] = …;
    signal(e_a[i]);
    …;
    wait(e_a[i-1]);
    … = a[i-1];
}
```

46

Example 1 (continued)

- Note that here it matters which iterations are assigned to which processor.
- It does not matter for correctness, but it matters for performance.
- Cyclic assignment is probably best.

47

Example 2: Enforcing Dependences

```
for( i=0; i<100; i++ ) a[i] = f(i);
x = g(a);
for( i=0; i<100; i++ ) b[i] = x + h( a[i] );
```

- First loop can be run in parallel.
- Middle statement is sequential.
- Second loop can be run in parallel.

48

Example 2 (contimued)

- We will need to make parallel execution stop after first loop and resume at the beginning of the second loop.
- Two (standard) ways of doing that:
  - fork() - join()
  - barrier synchronization

49

Fork-Join Synchronization

- fork() causes a number of processes to be created and to be run in parallel.
- join() causes all these processes to wait until all of them have executed a join().

50

Example 2 (continued)

```
fork();
for( i=...; i<...; i++ ) a[i] = f(i);
join();
x = g(a);
fork();
for( i=...; i<...; i++ ) b[i] = x + h( a[i] );
join();
```

51

Eliminating Dependences

- Privatization or scalar expansion
- Reduction (common pattern)

52

Example: Scalar Expansion or Privatization

```
for (I = 0; I < 100; I++)
    T = A[I];
    A[I] = B[I];
    B[I] = T;
```

Loop-carried anti-dependence on T
Eliminate by converting T into an array or by making T private to each loop iteration

53

Example: Scalar Expansion

```
for (I = 0; I < 100; I++)
    T [I]= A[I];
    A[I] = B[I];
    B[I] = T[I];
```

Loop-carried anti-dependence eliminated

54

Removing Dependences: Reduction

```
sum = 0.0;
for( i=0; i<100; i++ ) sum += a[i];
```

- Loop-carried dependence on sum.
- Cannot be parallelized, but ...

55

Reduction (continued)

```
for( i=0; i<...; i++ ) sum[i] = 0.0;
fork();
for( j=…; j<…; j++ ) sum[i] += a[j];
join();
sum = 0.0;
for( i=0; i<...; i++ ) sum += sum[i];
```

Common pattern often with explicit support
e.g., sum = reduce (+, a, 0, 100)
CAVEAT: Operator must be commutative and associative

56

Acknowledgements

Slides reflect content from Willy Zwaenepoel and from Grama/Gupta/Karypis/Kumar that accompany their corresponding course/textbooks and have been adapted to suit the content of this course