CSc 553

Principles of Compilation

36 : Parallelizing Compilers I

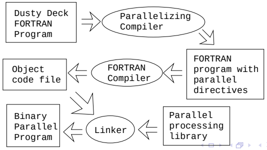
Department of Computer Science
University of Arizona

collberg@gmail.com

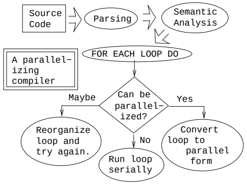
Parallelizing Compilers

Parallelizing Compilers I

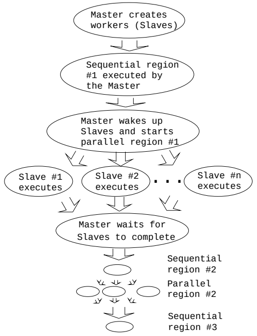
- Scientists/engineers want to speed up old FORTRAN programs by running them on multiprocessors. They don't want to rewrite the code to make explicit use of the available parallelism.
- Instead, they use **Parallelizing/Concurrentizing/Vectorizing** compilers that convert sequential programs to parallel form.


Dusty Deck FORTRAN Program
Parallelizing Compiler
FORTRAN program with parallel directives
FORTRAN Compiler
Object code file
Linker
Parallel processing library
Binary Parallel Program

Parallelizing Compilers II



- Concurrentizing compilers convert sequential programs to run on a shared memory multiprocessor. The resulting code uses a Master/Slave paradigm. The Master process executes sequential code (not all code can be parallelized), and starts up slave processors to execute (parts of) loops.



Master creates workers (Slaves)
Sequential region #1 executed by the Master
Master wakes up Slaves and starts parallel region #1
Slave #1 executes
Slave #2 executes
. . .
Slave #n executes
Master waits for Slaves to complete
Sequential region #2
Parallel region #2
Sequential region #3

Typical Concurrent Execution

Concurrentizing Compilers I

```
Sequential Program            ⟹    After parallelization
if x < 5 then                       # pragma sequential
   y := 17                          if x < 5 then
for i := 1 to 100 do                   y := 17
  A[i] := i*2                       # pragma parallel for-loop
end                                 for i := 1 to 100 do
x := 15                               A[i] := i*2
                                    end
                                    # pragma sequential
                                    x := 15
```

Concurrentizing Compilers II

Code for the Master

```
begin
   CreateSlaves(10);
   (* Slaves are now idle.  *)
   if x < 5 then y := 17
   for i := 1 to 10 do
      StartSlave(i, ForBody, (i − 1) ∗ 10 + 1, i ∗ 10)
   end
   WaitForSlaves();
   (* Slaves are idle again.  *)
   x := 15;
end

PROCEDURE ForBody(From, To :  INTEGER);
   for i := From to To do A[i] := i*2 end
END ForBody;
```

Concurrentizing Compilers III

- While the slaves are working, the Master is idle. This is of course wasteful, so normally the Master would schedule one set of iterations to itself.
- The Master could create a a new set of processes before each loop and then kill them off after the loop is finished. On many systems this would be expensive. Instead the Master starts the program by creating a set of processes and then "parks" them (i.e. puts them to sleep). When a parallel region is encountered, the Master wakes up the Slaves and hands them their workload.
- In the example, every Slave manipulates its own sub-array independently of the other processors. There are no cache conflicts. Often not this easy.

Prescheduled Loops I

- Here's a more detailed description of a **prescheduled loop**. A loop is prescheduled if each Slave is given enough information when it is awakened to execute all of its iterations of the loop.

```
WorkingSlaves := NoOfSlaves + 1;
(* The Master also works.  *)
InitMonitor(Lock);
ContinueAddress := LabelAfterLoop;
FOR i := 1 to NoOfSlaves DO
   From := ···; To := ···;
   StartSlave(i, ForBody, From, To);
END
ForBody(NoOfSlaves+1, ···);
LabelAfterLoop:
(* Master continues here.  *)
```

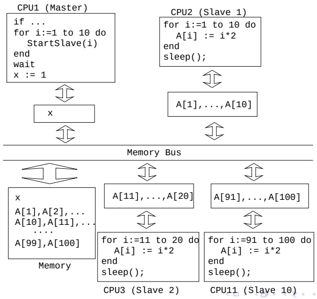
Prescheduled Loops II (a)

```
PROCEDURE ForBody (From, To :  int);
   FOR i := From to To DO
      A[i] := 2 * i;
   END;
   EnterMonitor(Lock);
      WorkingSlaves := WorkingSlaves - 1;
      IF WorkingsSlaves > 0 THEN
         ExitMonitor(Lock);
         WAIT;
      END;
   ExitMonitor(Lock);
   goto ContinueAddress;
END
```

Prescheduled Loops II (b)

- Note: This is **pseudo-code**. The actual implementation would obviously depend on what kind of support for thread creation & synchronization provided by the particular system.

Prescheduled Loops III



```
if ...
for i:=1 to 10 do
  StartSlave(i)
end
wait
x := 1
```

```
for i:=1 to 10 do
  A[i] := i*2
end
sleep();
```

```
x
A[1],A[2],...
A[10],A[11],...
   ....
A[99],A[100]
```

```
for i:=11 to 20 do
  A[i] := i*2
end
sleep();
```

```
for i:=91 to 100 do
  A[i] := i*2
end
sleep();
```

Prescheduled Loops IV

- Block-scheduled loops assign the same number of iterations to each processor. This is OK if all iterations perform the same amount of work.
- Consider this loop:

```
FOR i := 1 TO 1000 DO
   IF i > 500 THEN
      X := X + 1;
   ELSE
      FOR j := 1 to 100000 DO ··· END;
   END;
END
```

  If this loop was parallelized using block-scheduling, then some processors would be given a lot of work to do, others little. Hence, some processors will finish early, and won't contribute much to the processing.

Prescheduled Loops V

- **cyclic scheduling** is a variant of **block-scheduling** (which is what we've done so far).
- If we have N iterations to assign to P processes, block scheduling would assign $\lceil \frac{N}{P} \rceil$ consecutive iterations to each processor. I.e., process 1 would get iterations $1 \cdots \lceil \frac{N}{P} \rceil - 1$, process 2 iterations $\lceil \frac{N}{P} \rceil \cdots 2 \lceil \frac{N}{P} \rceil$, etc.
- Cyclic scheduling would instead assign every Pth iteration to each processor.

$P = 4$ processors $P_1 \cdots P_4$, $N = 12$ iterations

Block Scheduling				Cyclic Scheduling			
P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4
1	4	7	10	1	2	3	4
2	5	8	11	5	6	7	8
3	6	9	12	9	10	11	12

Self-Scheduled Loops I

- Prescheduled loops are **static**, i.e. the scheduling is decided at compile-time.
- **Self-scheduled** loops are **dynamic**, they assign workloads to processors at runtime. A dynamic scheduler assigns small amounts of work to each processor. When a processor finishes its task, it goes back to the scheduler and asks for more work.
- Problems:
  - There is overhead in the dynamic scheduler.
  - Should the scheduler hand out big chunks of work (⇒ less scheduling overhead; higher risk of load imbalance) or small chunks, e.g. single iterations (⇒ higher scheduling overhead; better load balance)?

Self-Scheduled Loops II

- In this example of a **self-scheduled loop**, there are `N` iterations to be scheduled. `CurrentIteration` holds the next iteration to be scheduled. Each Slaves tries to get work for itself, until all loops have been executed.

```
WorkingSlaves := NoOfSlaves + 1;
(* The Master also works.  *)
CurrentIteration := 1;
LastIteration := N;
ContinueAddress := LabelAfterLoop;
FOR i := 1 to NoOfSlaves DO
      StartSlave(i, ForBody);
END
ForBody();
LabelAfterLoop:
(* Master continues here.  *)
```

```
PROCEDURE ForBody ();
   LOOP
      EnterMonitor(Lock);
         i := CurrentIteration;
         IF i > LastIteration THEN EXIT;
         CurrentIteration ++;
      ExitMonitor(Lock);
      A[i] := 2 * i;
   ENDLOOP;
   WorkingSlaves := WorkingSlaves - 1;
   IF WorkingsSlave > 0 THEN
      ExitMonitor(Lock);
      WAIT;
   END;
   ExitMonitor(Lock);
   goto ContinueAddress;
END
```

Self-Scheduled Loops IV

- **Chunk Scheduling** removes some overhead of single-iteration scheduling:

```
CurrentIteration:=1; LastIteration:=N; ChunkSize:=10;
(* same as before *)
PROCEDURE ForBody ();
   LOOP
      EnterMonitor(Lock);
         i := CurrentIteration;
         IF i > LastIteration THEN EXIT;
         CurrentIteration += ChunkSize;
      ExitMonitor(Lock);
      FOR k := i TO i + ChunkSize DO
         A[k] := 2 * k; ENDFOR;
   ENDLOOP;
   (* same as before *)
END
```

Self-Scheduled Loops V

- The problem with **chunk scheduling** is that one Slave my wind up with a more expensive chunk than the others.
- **Guided Self-Scheduling** tries to alleviate this through **tapering**; the chunk-sizes get smaller towards the end of the loop. Specifically, the chunk-size is $\frac{\texttt{remaining iterations}}{2*\texttt{\# of processors}}$.

```
RemainingIters := N;
PROCEDURE ForBody ();
   LOOP i := CurrentIteration;
      ChunkSize := RemainingIters/(2*WorkingSlaves);
      RemainingIters -= ChunkSize;
      IF i > LastIteration THEN EXIT;
      CurrentIteration += ChunkSize;
      FOR k := i TO i + ChunkSize ···
   ENDLOOP;
```

Self-Scheduled Loops VI

- One problem with guided self-scheduling is the overhead of the scheduling operations themselves.
- **Factored Scheduling** computes the size of a **batch of chunks**. Each batch is a fraction $\frac{1}{\mathcal{F}}$ of the remaining work. Each chunk is $\frac{1}{\texttt{\# of processors}}$ of that batch.
- **Simple Factoring** sets $\mathcal{F} = 2$.

```
RemainingIters := N;
      ...
ChunkSize := RemainingIters/(WorkingSlaves * F);
RemainingIters := RemainingIters -
      WorkingSlaves * ChunkSize;
```

Self-Scheduled Loops VII

- Compare block scheduling, chunk self-scheduling, guided self-scheduling, and factoring when scheduling 100 iterations on 4 processors. Use chunk-size=10 and $\mathcal{F} = 2$.

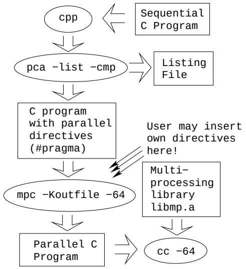
block	25 25 25 25
chunk	10 10 10 10 10 10 10 10 10 10
guided	13 11 10 9 8 7 6 5 4 4 3 3 3 2 2 2 2 1 1 1 1 1 1
factor	13 13 13 13 6 6 6 6 3 3 3 3 2 2 2 2 1 1 1 1

- Note that guided self-scheduling has 23 scheduling operations, factoring only 5 (since it schedules 4 chunks at a time).

A Parallelizing Compiler

A Parallelizing Compiler I

- SGI's **IRIX Power C** parallelizing compiler consists of two programs. $\boxed{\texttt{pca}}$ inserts parallel directives into the sequential C program. $\boxed{\texttt{mpc}}$ converts this program into parallel form. $\boxed{\texttt{cc}}$ compiles the resulting program.


cpp
Sequential C Program
pca -list -cmp
Listing File
C program with parallel directives (#pragma)
User may insert own directives here!
mpc -Koutfile -64
Multi-processing library libmp.a
Parallel C Program
cc -64

A Parallelizing Compiler II

- `/usr/lib/cpp mm.c > mm1.c`; `/usr/lib/pca -unroll=1 -cmp -list mm1.c`

```
#define N 600
double A[N][N],B[N][N],C[N][N];
main () {
   long i,j,k;
   for(i=1;i<=N; i++) {
      for(j=1;j<=N; j++) {
         A[i][j] = 0.0;
         for(k=1;k<=N; k++)
            A[i][j] = A[i][j]+B[i][k]*C[k][j];
         }
      }
}
```

```
double A[600][600],B[600][600],C[600][600];
main () {
   long i,j,k;
#pragma parallel shared(A, B, C)
                  local(i, j, _Kdd1, k)
   {
#pragma pfor iterate(i=1;600;1)
      for ( i = 1; i<=600; i++ )
         for ( j = 1; j<=600; j++ )
            A[i][j] = 0.e0;
#pragma synchronize
#pragma pfor iterate(i=1;600;1)
   for ( i = 1; i<=600; i++ )
      for ( k = 1; k<=600; k++ ) {
         _Kdd1 = B[i][k];
         for ( j = 1; j<=600; j++ )
            A[i][j]=A[i][j]+_Kdd1*C[k][j];
      }
   }
}
```

A Parallelizing Compiler IV

- pca analyses the C-code and inserts #pragmas which are then used by `mpc`.
- `#pragma parallel` starts a parallel region. `#pragma pFor` starts a parallel for loop. `#pragma synchronize` is a classic **barrier** construct.
- The next slide shows the explanation produced by the compiler.

A Parallelizing Compiler IV

```
for i
    Line:8  Unrolling of this loop was not
    done because size is ok asis.
    Original loop split into sub-loops
    1. Concurrent
    2. Concurrent
for j
    Original loop split into sub-loops
    1. Scalar Already in a parallel loop.
    2. Scalar Already in a parallel loop.
for k
    1. Scalar Already in a parallel loop.
Optimization Summary
             1 loops concurrentized
             2 preferred scalar mode
```

Summary

Summary I

- Read:

  Concurrentization: Michael Wolfe, *High Performance Compilers for Parallel Computing*, pp. 385–392.

- A concurrentizing compiler should
  1. find loops that can be parallelized,
  2. reorganize remaining loops so that they can also be parallelized,
  3. devise the best scheduling policy for each loop. We've seen

     static (pre) scheduling: block, cyclic

     dynamic (self) scheduling: single iteration, chunk, guided, factoring,
  4. explain to the user why some loops could not be parallelized.

Summary II

- Static scheduling works well when the cost of each iteration can be determined at compile time. This is true of most scientific codes since these typically perform simple arithmetic on each element of an array.
- Dynamic scheduling handles loops where the cost of each iteration can't be determined at compile time. Usually this is because the loop contains an `IF`-statement or a function call.
- Most concurrentizing compilers only parallelize **outermost** loops.

Summary III

- Compilers for parallel vector machines (such as the Cray J916) will concurrentize outer loops and vectorize inner loops. In the loop-nest below, for example, the `j` loop would be converted to vector operations, and the `i` loop would be scheduled over the available processors:

```
FOR i := 1 TO 1000 DO
   FOR j := 1 TO 64 DO
      A[i,j] := A[i,j]*2
```

Homework

Homework I

- Compare block scheduling, chunk self-scheduling, guided self-scheduling, and factoring when scheduling 120 iterations on 6 processors. Use chunk-size=12 and $\mathcal{F} = 2$.

block	
chunk	
guided	
factor	