ParaWise/CAPO Parallelization Environment

Henry Jin
hjin@nas.nasa.gov
NASA Advanced Supercomputing (NAS) Division
NASA Ames Research Center
June 13, 2005
Key Ideas

- Interactive environment for semi-automatic parallelization of Fortran application codes
- Generated codes in recognizable form by user

INPUT

Fortran code

ParaWise/CAPO

Transformation
Parallel code

OUTPUT

Fortran + OpenMP directives
ParaWise and CAPO

• ParaWise
  – Semi-automatic, developed by Parallel Software Products
  – Accurate symbolic, value based, interprocedural data dependence analysis
  – Domain decomposition for generating message-passing codes
  – A set of browsers for user to interact with the parallelization process

• CAPO
  – A module for generating OpenMP parallel codes, developed at NASA Ames
  – Exploits loop-level parallelism
  – Directives browsers to guide the parallelization process
  – Currently integrated with ParaWise
Interactive Parallelization Process

1. Source
2. Dependence Analysis
3. Directive Analysis
4. OpenMP Code Generation
5. Parallel Code

- Loop Analysis
- Parallel Region Creation
- Region Merging and Migrating
- NOWAIT Optimization
- THREADPRIVATE Analysis
- Variable Scoping Analysis
- Code Transformation
- Directive Insertion
Generation of OpenMP Code

• Identify parallel loops, including loops for setting up possible pipeline

• Construct parallel regions from parallel loops

• Merge consecutive parallel regions and migrate parallel regions as high as possible in the call path

• Perform NOWAIT optimization for consecutive parallel loops inside a parallel region

• Automatically identify and define variable scopes, such as SHARED, PRIVATE and REDUCTION

• Detect and produce THREADPRIVATE directives for common blocks
Code Generation Process

serial code

do K=
  ...
end do

call subwork
  ...

subroutine subwork
  do J=
    ...
  end do

do J=
    ...
end do

return
end

identify parallel loops
create parallel regions

!$OMP PARALLEL DO
  do K=
    ...
  end do
$OMP END PARALLEL DO

call subwork
  ...

subroutine subwork
  ...

!$OMP PARALLEL DO
  do J=
    ...
  end do
$OMP END PARALLEL DO

!$OMP PARALLEL DO
  do J=
    ...
  end do
$OMP END PARALLEL DO

return
end
Code Generation Process (cont.)

merge parallel regions

migrate parallel regions
generate NOWAIT
Automatic Code Transformation

• Privatization of common block variables
  – if cannot be handled with THREADPRIVATE

• Routine duplication
  – to resolve conflicts of usage

• Reduction on an array variable
  – update local variable in parallel, then the shared array variable in a critical region

• F90 array syntax to loop nest
  – so that OMP DO can be applied

• Loop interchange
  – for better cache utilization
Routine Duplication

- Call inside a parallel region, but not inside a parallel DO

```fortran
!$OMP PARALLEL
  call cap_sub
!$OMP DO
    do K=
      ...
    end do
  !$OMP END PARALLEL
  ...
  call cap_sub
!

subroutine cap_sub
!

!$OMP PARALLEL DO
  do J=
    ...
  end do
!$OMP END PARALLEL
  ...
  call sub
!

subroutine sub
!
```

inside parallel region

outside parallel region
Identifying Parallel Loops - The Key Issue

• Code developers want to
  – find all the loops that can be parallelized
  – find all those that look ‘serial’
  – find which of the ‘serial’ don’t affect parallel performance and which are critical
  – fix the code so that the critical ‘serial’ loops can be parallelized

• CAPO enables this function by
  – categorizing different loop types
  – solving through user interaction
  – generating parallel code with directives automatically
Directives Browser Window

Loop Filter:
- Totally Serial
- Covered Serial
- Chosen Parallel
- Not Chosen

Sub Filter:
- All
- True Recursion
- Privatization
- I/O or Exit
- Inside Parallel
- User Defined

More Filter:

Current Routine: SOLVEP

Loop:
- DO 331 K=KS,KE,1
- DO 331 J=1,JE,1

IO/Exit statements:
- Contains 15 parallel loops
- 5 variables with loop-carried true dependence (level=4)
- 12 variables with loop-carried anti dependence (level=4)
- 14 variables with loop-carried output dependence (level=4)
- and non-privatizable, due to usage from outside the loop

Contains 15 parallel loops:

Inside parallel loops:
- SOLVEP: DO 301 i=1,IT,1
- SOLVEP: DO 301 i=1,IT,1
- SOLVEP: DO 301 i=1,IT,1
- SOLVEP: DO 12,IT,1
- SOLVEP: DO 12,IT,1

Hints:
- Contains 15 parallel loops
- 5 variables with loop-carried true dependence (level=4)
- 12 variables with loop-carried anti dependence (level=4)
- and non-privatizable, due to usage from outside the loop

More Browsers:
- Region...
- Array Syntax...
- Routine Dup...

Routine Dup...
Totally Serial

Problem: Potentially severe
- Serial due to loop-carried true dependence present and/or,
- Serial due to loop-carried pseudo (memory re-use) dependence by a non-privatizable variable
- Not contained in, or containing ANY parallel loops - entirely serial
- Sequential execution can prevent effective parallel performance

Possible Solutions:
- True dependence may have been assumed, may be proven to no longer exist if user knowledge is added.
- Investigate loop-carried pseudo dependence - add user knowledge to prove non-existence.
- Investigate privatization preventing true dependences from/to outside of loop - add user knowledge to prove non-existence

Browser shows serializing dependences (textually and graphically)
Covered Serial

*Problem: May be important*

- Also a serial loop, but contains or is contained in a parallel loop so some parallelism will be exploited.
- If contains parallel loops, parallel performance can be enhanced by parallelism at this higher level.

*Possible solutions:*

- Can be treated in a similar manner to the “serial” loop type described previously.

Browser shows serializing dependences and surrounding parallel loop(s) and/or contained parallel loops
Loop Types Identified with Directives Browser (cont.)

Chosen Parallel:
- Parallel loop that is not nested within other parallel loops
- Current Loop level at which parallel DO directive is inserted
- Includes loops identified with reduction operations
- Includes loops identified with software pipelines

Not Chosen:
- Parallel loop not chosen due to the selection of other parallel loops from the “Chosen Parallel” category above or due to I/O statements
- User may enforce parallelization if needed
The Why Directives Window

- Reason and hints for a selected loop
- List of variables and dependence types
- Tools for removing dependences
Investigate Why a Dependence Is Defined
Further Code Optimization

• Choose outer-most loops for better granularity

• Prune data dependences when
  – unknown information involved (e.g. input parameters)
  – code too complicated (e.g. FFT)

• Require user knowledge

• Use dialog boxes in the WhyDirectives window
  – remove *false* data dependences
  – thus parallelize a loop
integer indexptr(maxcells)
read*,indexptr
do i=1,ncells
  $1\ u(indexptr(i)) = \ldots$
  $2\ \ldots = u(indexptr(i)) + \ldots$
enddo
$3\ print*,(u(j),j=1,ncells)$

**analysis**
- i loop serial due to loop carried pseudo dependences of $u$, $S_1 \rightarrow S_1$ (output), $S_2 \rightarrow S_1$ (anti), Loop output $S_1 \rightarrow S_3$ also $u$ is not PRIVATE

**user inspection**
- Examine Loop output dependence and determine it is correct therefore $u$ cannot be PRIVATE

**possible solution**
- If contents of indexptr are all unique then we can safely remove the loop carried anti and output dependencies for the array $u$ allowing $u$ to stay SHARED and the loop to execute in parallel
Remove Data Dependences (cont.)

\[ S_1 \quad \text{read}^*, (\text{work}(k), k=1,10) \]
\[ \quad \text{do } i=1,10 \]
\[ \quad \quad \text{do } j=1,n \]
\[ S_2 \quad \text{work}(j)=j \]
\[ \quad \text{enddo} \]
\[ S_3 \quad \text{b}(i)=\text{b}(i)+\text{work}(2) \]
\[ \quad \text{enddo} \]

**analysis**
- \( i \) loop serial due to loop carried pseudo dependencies of \( \text{work} \), \( S_2 \rightarrow S_2 \) (output), \( S_3 \rightarrow S_2 \) (anti)
- Loop input dependence of \( \text{work} \), \( S_1 \rightarrow S_3 \) (true) exists so \( \text{work} \) is not PRIVATE

**user inspection**
- Examine in Why dependence window of dependence graph browser
- Determine that the pseudo dependencies are correct (work is re-used)
- Loop input dependence non-existent if \( n \geq 2 \)

**possible solution**
- Delete loop input dependence or (preferably) add \( n \geq 2 \) to info + re-analyze. work is now PRIVATE and \( i \) loop can execute in parallel
Remove Data Dependences (cont.)

\[ S_1 \quad \text{read*, (work(k),k=1,10), (n(k),k=1,10)} \]
\[ \text{do } i=1,10 \]
\[ \quad \text{do } j=1,n(i) \]
\[ S_2 \quad \text{work(j)=j} \]
\[ \quad \text{enddo} \]
\[ S_3 \quad b(i)=b(i)+\text{work(2)} \]
\[ \quad \text{enddo} \]

\textit{analysis}

- Now \( n \) is an array – additional \textit{true} dependence of work carried by \( i \) loop \( S_2 \rightarrow S_3 \)
- \( i \) loop appears to be inherently serial

\textit{user inspection}

- Examine \textit{true} dependence first, others only important if it can be removed
- Loop carried \textit{true} dependence non-existent if all \( n(1:10) \geq 2 \)

\textit{possible solution}

- Delete loop carried \textit{true} dependence followed by loop input dependence (as before) or just add \( n(1:10) \geq 2 \) to info + re-analyze
- \( i \) loop is now parallel and \texttt{work} is \texttt{PRIVATE}
Remove Data Dependences (cont.)

do k=1,d(3)
  do j=1,d(2)
    do i=1,d(1)
      y1(j,i)= . . .
    enddo
  enddo
S_1
S_2  call cfftz(y1, . . .)
  do j=1,d(2)
    do i=1,d(1)
      . . . = y1(j,i)
    enddo
S_3

analysis
• k loop is apparently serial since $y_1$ is assigned in $S_1$ and $S_2$ and is used in $S_2$ and $S_3$ i.e. true dependence $S_2 \rightarrow S_2$

user inspection
• Examine true dependence first, others only important if it can be removed.
• Examine loop Input/Output dependence

possible solution
• If it is known that there are no assignments of $y_1$ before $S_1$ then we can safely remove the loop carried true dependences and Input/Output dependences for $y_1$ making it PRIVATE
Remove Data Dependences (cont.)

• **[DeLoop]**
  - make variables *shared* → delete *loop-carried* dependences

• **[Privatize]**
  - make variables *private* → delete loop-carried *True/Anti* dependences and *Input/Output* True dependences

![CAPO: VarList DeLooping](image1)

![CAPO: VarList Privatization](image2)

- Click Apply to remove Loop-Carried Dependences for the selected variables!
- Click Apply to perform Privatization for the selected variables!
Remove Data Dependences (cont.)

- [Privatize] continued
  - It is possible to make variables firstprivate or lastprivate → select “Remove Output (> ) or Input (< ) dependences”

Caution! User can improve performance but also can introduce mistakes
Further Optimization

• User enforced loop types
  – overwrite a default
    • for I/O loops
    • concerning granularity
  – use the Loop Type window

• The “userloop.par” file
  – User defined loop types are saved to this file, read back automatically from the file
  – A different filename may be specified via the environment variable CAPO_USERLOOP
Parameters to Control the CAPO Execution

- Setting dialog box
  - set most parameters

- Environment variables
  - GUI correspondence
    - CAPO_LOGINFO
    - ...
  - no GUI correspondence
    - CAPO_USERLOOP
    - ...

if you are not sure
Browsing Parallel Regions

- With the *Parallel Regions* browser
Browsing Parallel Regions (cont.)

- Connection to the WhyDirectives window
  - list of variables and their types
  - indication of the end-of-loop synchronization
- No direct modification to regions
Hotlinks

- Quick access to other functions
- Menus from pressing the *right* mouse button
  - linked with a loop
  - linked with a variable
  - linked with a routine
  - linked with a textline
- Example
  - bring up the DepGraph window for the selected loop
Command Interface for the Batch Mode

- Provide access to the functionality of GUI components without starting the GUI
- Commands usually recorded to a command file by
  ```
  capo -logfile capo_run.cmd
  ```
- Played back [in a batch mode] with
  ```
  capo [-batch] capo_run.cmd
  ```
- Commands in the command interface are given in the CAPO User Manual A4
Hybrid Parallelization

- Existing message passing codes
  - Use CAPO to insert OpenMP directives

- Two-step process
  - First: ParaWise to generate the message-passing code
  - Second: CAPO to insert OpenMP directives

- Issues
  - No communication routines allowed inside a parallel region
  - The partitioned dimension is not used for OMP loop level parallelization, but it is possible to enforce the choice
    - In the Setting Box, check “Use Partitioned Loop”

- See an example in the CAPO tutorial notes
Fortran 90/95 Codes

- In the beta stage
- Main feature – handling array syntax, **FORALL** loop, **WHERE** construct
  - convert to a regular DO loop
  - use "**OMP WORKSHARE**" (not yet supported)
  - do nothing, let a compiler work it out

```
flux(2,2:nx-1,2:ny-1,2:nz)=tz3*(du2(2:nx-1,2:ny-1,2:nz) &
&                            -du2(2:nx-1,2:ny-1,1:nz-1))
```

converted to

```
do  ARRAY_VAR3=2,nz
    flux(2,1:nx,2:ny-1,ARRAY_VAR3)=tz3* &
    (du2(2:nx-1,2:ny-1,ARRAY_VAR3) &
     & -du2(2:nx-1,2:ny-1,ARRAY_VAR3-1))
end do
```
Fortran 90/95 Codes (cont.)

- Control of the array syntax conversion
  - done automatically
  - user can overwrite:
    select an index dimension for conversion