Automatic Data Distribution Method for Distributed Shared Memory Machines

Advanced Parallelizing Compiler Project
Systems Development Laboratory, Hitachi,Ltd.

Takashi HIROOKA

March 20, 2003

Contents

1.1 Background

The clock speed of a single CPU is growing rapidly.

However, it will run up against an *atomic* wall in the near future.

So, various types of multiprocessors are proposed:

- SMP: Shared Memory Multiprocessors
- DMP: Distributed Memory Multiprocessors
- DSM: Distributed Shared Memory Multiprocessors

The DSM inherits both advantages of SMP (easy parallel programming) and DMP (good scalability).

So, it is promising.

1.2 Disadvantage of DSM & Conventional methods

One of disadvantages of DSM is

slow remote memory accesses.

The key solution to overcome it is

to increase data locality

(or to distribute data appropriately).

Some methods are proposed:

- **First Touch mechanism**
- **Page Migration**
- **Data Distribution Directives**

1.3 Conventional methods (Definition)

- **First Touch data distribution mechanism**

  OSs assign each page to the node that accesses the page first.

- **Page Migration mechanism**

  OSs migrate a page dynamically to the node that accesses the page frequently.

- **Data distribution directives**

  Users insert those directives in their programs. Then, a compiler assigns the specified data to the nodes in the specified manner.

1.4 Problem & Purpose

Features & problem

Method	First Touch (OS)	Page Migration	Directives
Features	Good for consistent access pattern	Dynamic, flexible	Good for consistent & regular pattern
Problem	It depends on the 1st access pattern	Large overhead	Difficult to specify ir-regular pattern

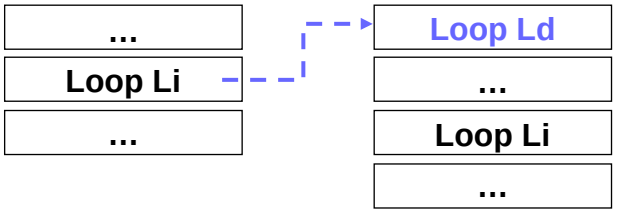
Purpose

To propose a data distribution method that
is independent of the 1st acces pattern,
has low overhead, and
supports irregular access patterns.

➡ First Touch Control by Compiler

2.1 Basic Strategy of First Touch Control Method

(1) Find in a program the most important loop (Li);
i.e., that consumes much execution time.
(2) Extract a data access pattern in the loop.
(3) Make a loop (Ld) realizing the pattern.
(4) Place the loop at the beginning of the program.



The FTC is independent of the 1st acces pattern, has low overhead (only assignment stmt), and supports irregular access patterns.

2.2 Example - Array section reference – (1)

Example program

```
1:      real A(100)

21: c Initialization loop
22: !$omp parallel do
23:       do i=1, 100
24:         A(i)= 0
25:       enddo
          ' ' '
31: c Kernel loop
32:       do itr=1, 10000
33: !$omp parallel do
34:         do i=41, 100
35:           A(i)= ...
36:         enddo
37:       enddo
```

Inserted in the declaration part

(b) Conventional method

```
11: c Data distribution directive
12: c$distribute A(block)
```

Place the loop at the beginning of the program.

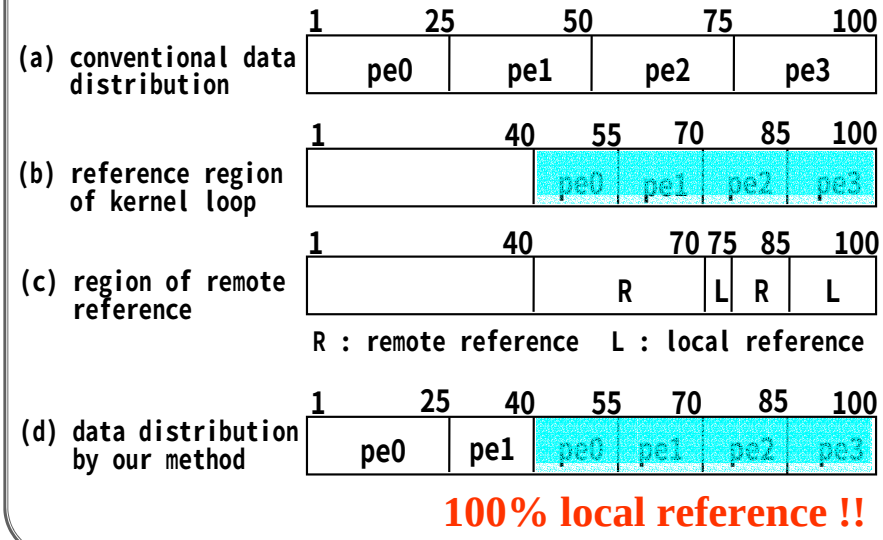
(c) First touch control method

```
11: c FTC loop
12: !$omp parallel do
13:       do i=41, 100
14:         A(i)=0
15:       enddo
```

Make a loop realizing the reference pattern of A.

2.3 Example - Array section reference – (2)

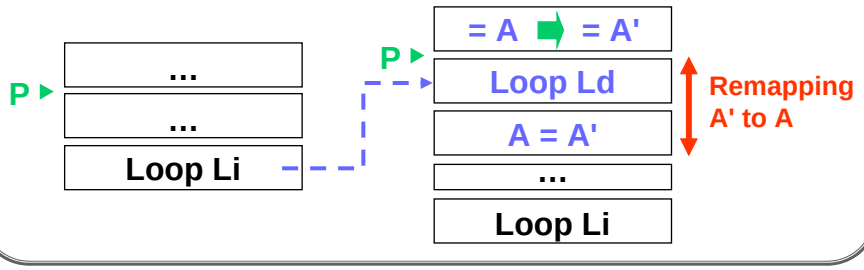
Illustration of data distribution



100% local reference !!

3.1 FTC Method to indirect reference arrays

(1) Find in a program the most important loop (Li) that includes indirect references to A(J(i)).
(2) Extract a data access pattern in the loop.
(3) Make a loop (Ld) realizing the pattern.
(4) Place the loop just after the point (P) where the values of the index array (J) are calculated.
(5) Replace each reference to A preceding P with A'.
(6) Insert A=A' following Ld.



3.2 Example - NPB2.3serial / CG -

Our FTC method is applied interprocedurally.

Kernel loop of CG

```
!$omp parallel do
do j = 1, n
  do k = rowstr(j), rowstr(j+1)-1
    sum = sum + a(k) * p(colidx(k))
  enddo
  q(j) = sum
enddo
```

Call graph of CG

```
main +makea +[Initialization loop for colidx_clone]
       .....
     |    +sparse +[Initialization loop for a']
     |    |       +[value of rowstr is calculated]
     |    |       +[FTC loop for a]
     |    |       +[a = a' ]
     |    +[FTC loop for colidx]
     |    +[colidx = colidx']
     +conj_grad (Kernel loop)
```


4.1 Evaluation Environment

Evaluation System:

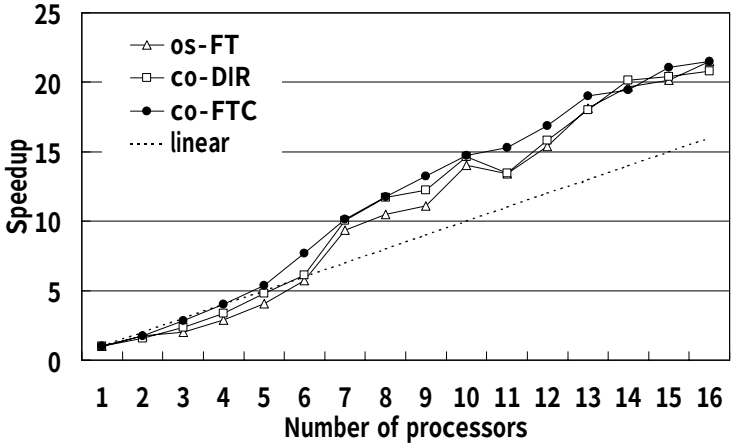
SGI® Origin® 2000

- **CPU: R10000®, 195MHz**
- **Structure: DSM (32 PEs= 2PEs/node x 16 nodes)**
- **L1: 32KB / 32KB**
- **L2: 4MB/PE**
- **Compiler: MIPSpro™ Fortran90 Version 7.30**

Evaluation Programs:

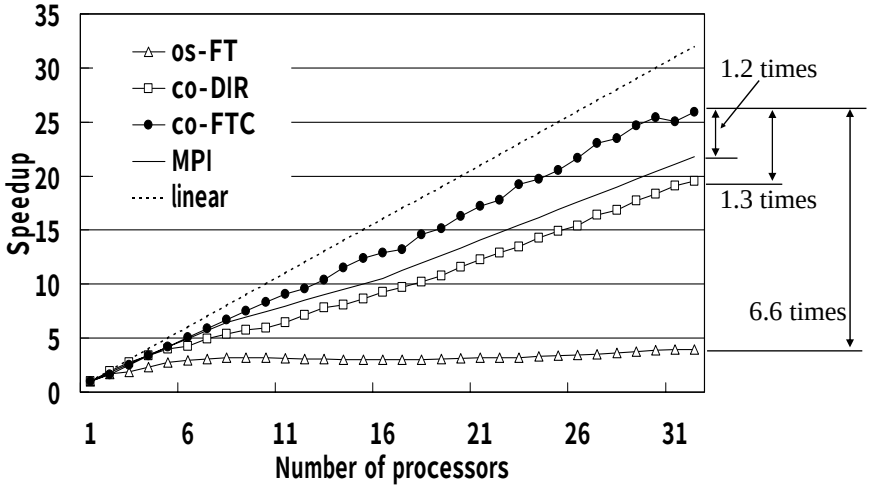
os-FT: original
co-DIR: compiler data distrib. directive
co-FTC: proposed FTC method

4.1 Speedup of CG (class A)



os-FT: original
co-DIR: add data distrib. directive
co-FTC: proposed FTC method

4.2 Speedup of CG (class B)



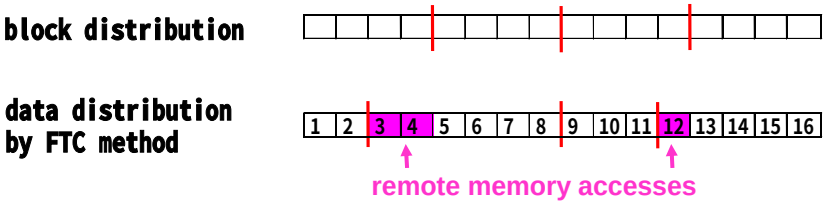
os-FT: original, co-DIR: add data distrib. directive,
co-FTC: proposed FTC method

4.3 Illustration of data distribution

Kernel loop of CG

```
!$omp parallel do
do j = 1, n
  do k = rowstr(j), rowstr(j+1)-1
    sum = sum + a(k) * p(colidx(k))
  enddo
  q(j) = sum
enddo
```

Illustration of data distribution



5. Conclusions

- **We have implemented our automatic data distribution method for DSM.**
- **On SGI® Origin® 2000 (32 processors), the FTC-version of NPB2.3serial/CG (class B) ran 6.6 times faster than the conventional versions.**