

Programming Parallel Machines

Prof. R. Eigenmann

ECE563, Spring 2013

engineering.purdue.edu/~eigenman/ECE563

Parallelism - Why Bother?

- ◆ Hardware-Perspective: Parallelism is everywhere
 - instruction level
 - chip level (multicores)
 - co-processors (accelerators, GPUs)
 - multi-processor level
 - multi-computer level
 - distributed system level

Big Question: Can all this parallelism be hidden ?

Hiding Parallelism - For Whom and Behind What?

- ◆ For the end user:
 - HPC applications: parallelism is not really apparent. Sometimes, the user needs to tell the system on how many “nodes” the application shall run.
 - Collaborative applications and remote resource accesses: the user may *want* to see the parallelism.
- ◆ For the application programmer:

We can try to hide parallelism

 - using parallelizing compilers
 - using parallel libraries or software components
 - In reality: partially hide parallelism behind a good API

Different Forms of Parallelism

- ◆ An operating system has parallel processes to manage the many parallel activities that are going on concurrently.
- ◆ A high-performance computing application executes multiple parts of the program in parallel in order to get the job done faster.
- ◆ A bank that performs a transaction with another banks uses parallel systems to engage multiple distributed databases
- ◆ A multi-group research team that uses a satellite downlink in Boston, a large computer in San-Diego and a “Cave” for visualization at the Univ. of Illinois needs collaborative parallel systems.

How important is Parallel Programming 2013 in Academia?

- ◆ It's a hot topic again
 - There was significant interest in the 1980es and first half of 1990es.
 - Then the interest in parallelism declined until about 2005.
 - Increased funding for HPC and emerging multicore architectures have made parallel programming a central topic.
- ◆ Hard problems remain:
 - what is the right user model and application programmer interface (API) for high performance and high productivity ?
 - what are the right programming methodologies?
 - how can we build scalable hardware and software systems?
 - how can we get the user community to accept parallelism?

How important is Parallel Programming 2013 in Industry?

- ◆ **Parallelism has become a mainstream technology**
 - The “multicore challenge” is one of the current big issues.
 - Industry’s presence at “Supercomputing” is growing.
 - Most large-scale computational applications exist in parallel form (car crash, computational chemistry, airplane/flow simulations, seismic processing, to name just a few).
 - Parallel systems sell in many shapes and forms: HPC systems for “number crunching”, parallel servers, multi-processor PCs.
 - Students who learn about parallelism find good jobs.
- ◆ **However, the hard problems pose significant challenges:**
 - Lack of a good API leads to non-portable programs.
 - Lack of a programming methodology leads to high software cost.
 - Non-scalable systems limit the benefit from parallelism.
 - Lack of acceptance hinders dissemination of parallel systems.

What is so Special About Parallel Programming?

Parallel programming is hard because:

- ◆ You have to “think parallel”
 - We tend to think step-by-step, which is closer to the way a sequential program is written.
- ◆ All algorithms have ordering constraints
 - They are a.k.a. data and control dependences, and they are difficult to analyze and express => you need to coordinate the parallel threads using synchronization constructs (e.g., locks)
- ◆ Deadlocks may occur.
 - Incorrectly coordinated threads may cause a program to “hang”.
- ◆ Race conditions may occur
 - Race conditions occur when dependences are violated. They lead to non-reproducible answers of the program. Most often this is an error.
- ◆ Data partitioning, off-loading, and message generation are tedious
 - But unavoidable when programming distributed and heterogeneous systems.

So, Why Can't We Hide Parallelism?

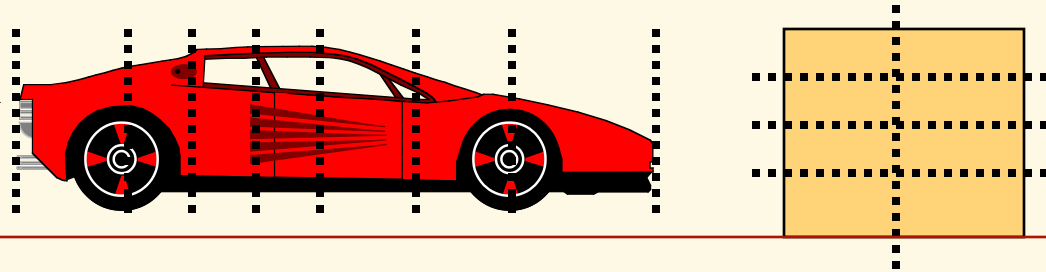
- ◆ Could we reuse parallel modules?
 - Software reuse is still largely an unsolved problem
 - Parallelism encoded in reusable modules may not be at the right level
 - ▼ Parallelizing an inner loop is almost always less efficient than an outer loop.
- ◆ Could we use parallelizing compilers?
 - Despite much progress in parallelizing compilers, there are still many programs that fail to use such tools, e.g.
 - ▼ programs not written in Fortran77 and C
 - ▼ irregular programs (e.g., using sparse data structures)

Basic Methods For Creating Parallel Programs

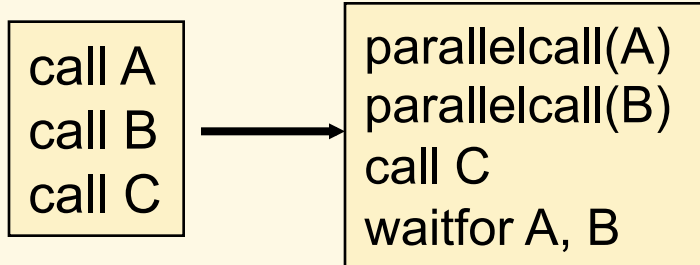
- ◆ Write a sequential program, then parallelize it.
 - **Application-level approach:** Study the physics behind the code. Identify steps that can be executed concurrently. Re-code these steps in parallel form.
 - **Program-level approach:** Analyze the program. Find code sections that access disjoint data. Mark these sections for parallel execution. *Our focus*
- ◆ Write a parallel program directly (from scratch)
 - Advantage: parallelism is not inhibited by the sequential programming style.
 - Disadvantages:
 - ▼ Large, existing applications: huge effort
 - ▼ If sequential programming is difficult already, this approach makes programming even more difficult

Where in the Application Can We Find Parallelism?

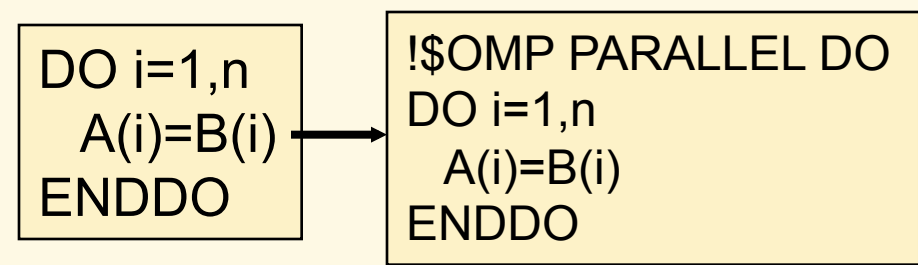
Coarse-grain parallelism through domain decomposition



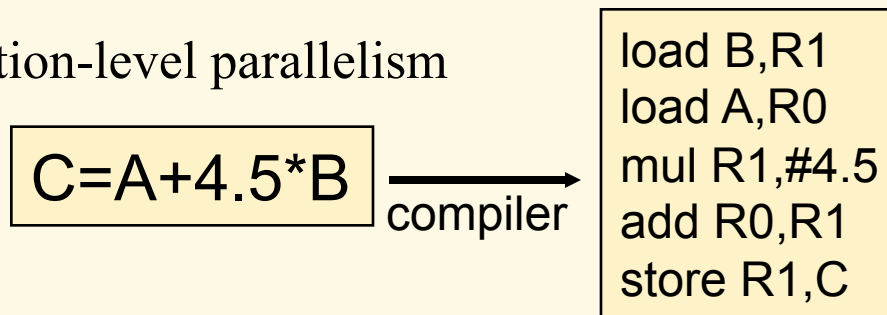
Subroutine-level parallelism



Loop-level parallelism



Instruction-level parallelism

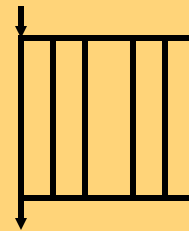


How Can We Express This Parallelism?

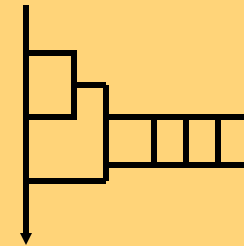
Parallel Threads

```
new_thread(id,subr, param)
```

```
wait_thread(id)
```



all threads live from the beginning to the end of the program or program section



thread creation dynamic, as needed.

Parallel Loops

```
PARALLEL DO i=1,n  
  call work(i)  
ENDDO
```

outermost loop in program is parallel

```
DO i=1,n
```

```
  PARALLEL DO i=1,n  
  ...  
  ENDDO
```

```
  PARALLEL DO i=1,n  
  ...  
  ENDDO
```

```
  PARALLEL DO i=1,n  
  ...  
  ENDDO
```

```
ENDDO
```

multiple inner parallel loops

How Can We Express This Parallelism?

Parallel Sections

```
COBEGIN
  task1
||
  task2
||
  task3
COEND
```

task 1, 2, and 3 are executed concurrently

SPMD execution

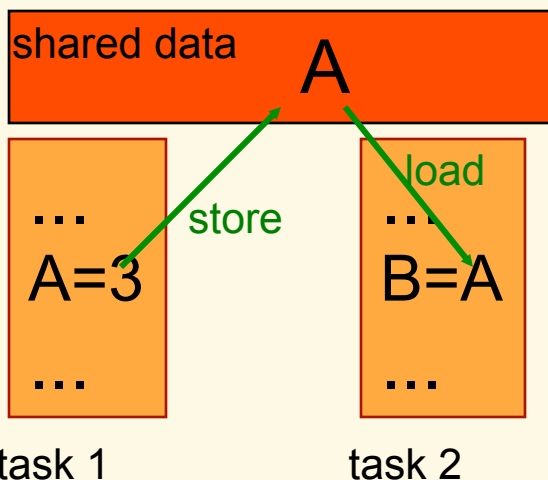
May be implicit

```
BEGIN (all in parallel)
  ...do-the-work...
END
```

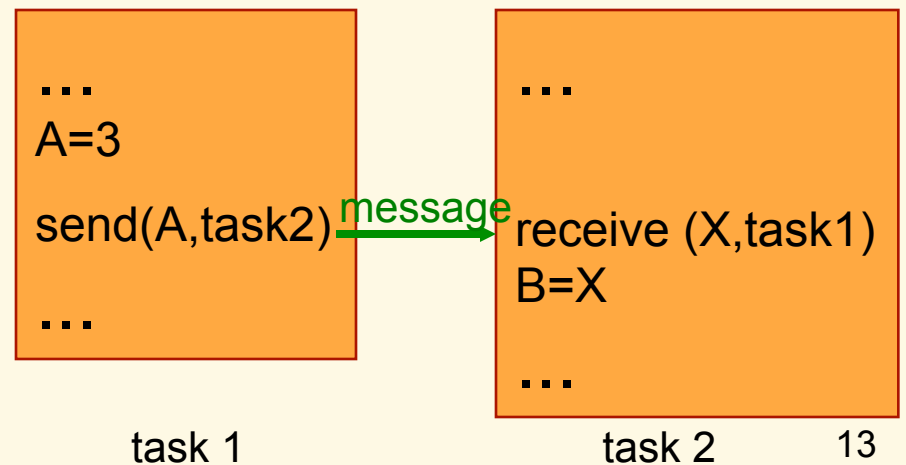
a copy of "do-the-work" is executed by each processor/thread

How do Parallel Tasks Communicate?

shared-address-space,
global-address-space, or
shared-memory model:
tasks see each other's data
(unless it is explicitly
declared as private.)



distributed-memory or
message-passing model:
tasks exchange data
through explicit
messages



Generating Parallel Programs Automatically

A quick tour through a
parallelizing compiler

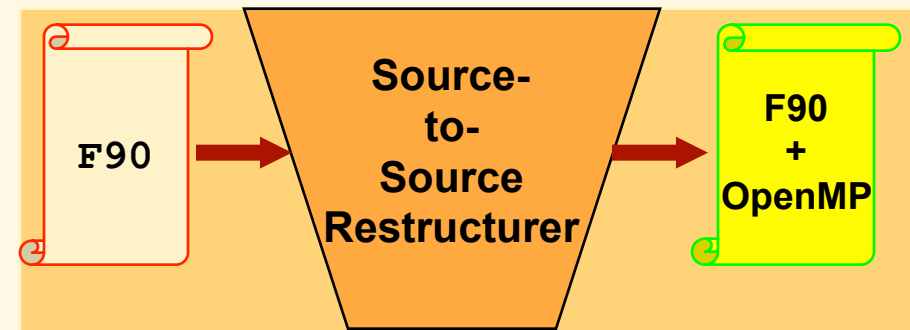
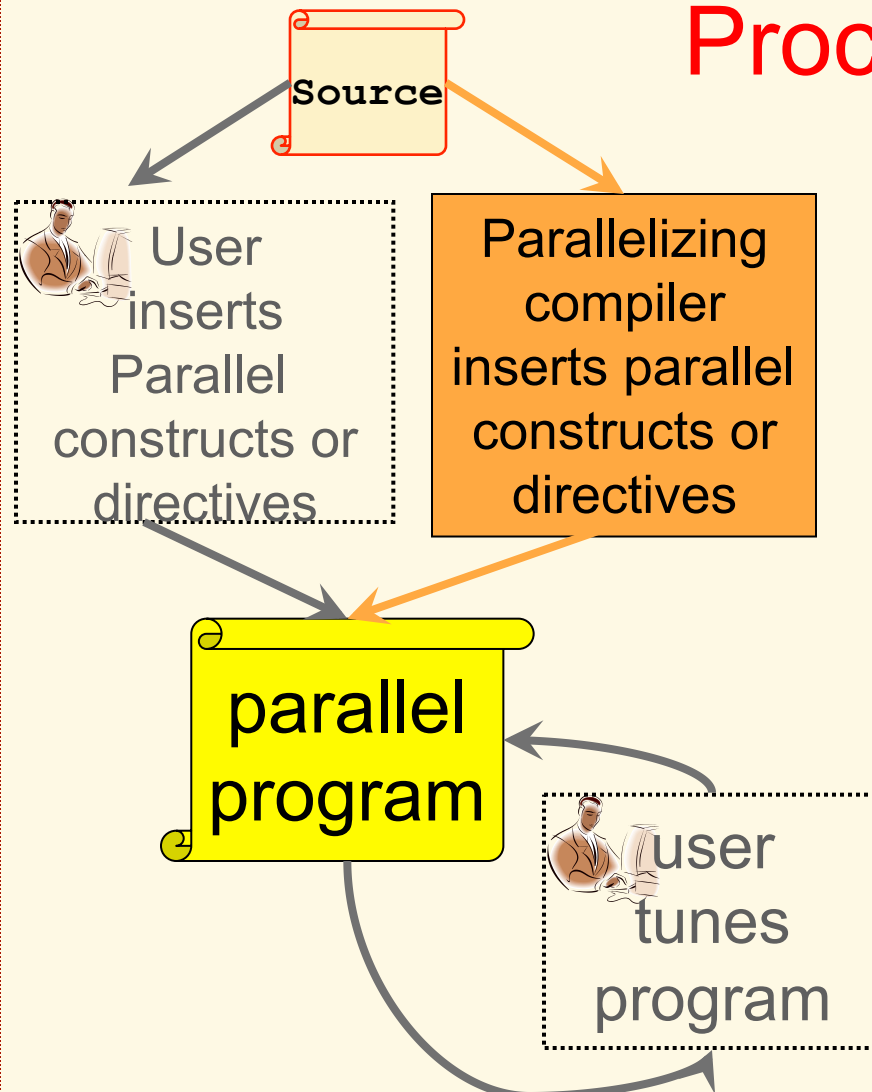
*The important techniques of parallelizing
compilers are also the important techniques for
manual parallelization*

See *ECE 663 Lecture Notes and Slides*
engineering.purdue.edu/~eigenman/ECE663/Handouts

Automatic Parallelization

- ◆ The Scope of Auto-Parallelization
- ◆ Loop Parallelization 101
- ◆ Most Influential Dependence Removing Transformations
- ◆ User's Role in "Automatic" Parallelization
- ◆ Performance of Parallelizing Compilers

Where Do Parallelization Tools Fit Into the Software Engineering Process ?



Source-to-source restructurers:

- F90 → F90/OpenMP
- C → C/OpenMP

examples:

- SGI F77 compiler (-apo -mplist option)
- Cetus compiler

The Basics About Parallelizing Compilers

- ◆ Loops are the primary source of parallelism in scientific and engineering applications.
- ◆ Compilers detect loops that have independent iterations, i.e. iterations access disjoint data

```
FOR I = 1 TO n
  A[expression1] = ...
  ... = A[expression2]
ENDFOR
```

Parallelism of loops
accessing arrays:

The loop is independent if, *expression1* is different from *expression2* (for any two different iterations)

Parallel Loop Execution

- ◆ Fully-independent loops are executed in parallel.

```
FOR i=1 TO 250
...
ENDFOR
```

```
FOR i=251 TO 500
...
ENDFOR
```

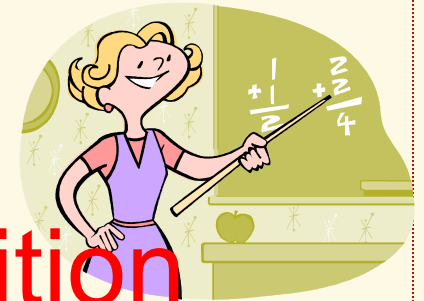
```
FOR i=501 TO 750
...
ENDFOR
```

```
FOR i=751 TO 1000
...
ENDFOR
```

The execution of this loop on 4 processors may assign 250 iterations to each processor

- ◆ Usually, any dependence prevents a loop from executing in parallel. Hence, removing dependences is very important.

Loop Parallelization 101



Data Dependence: Definition

A dependence exists between two data references if
(in a sequential execution of the program)

- both references access the same storage location,
- **and** at least one of them is a write reference.

Basic data dependence classification:

Read after Write (RAW): flow dependence	true dependence
Write after Read (WAR): anti dependence	false, or storage- related dependences
Write after Write (WAW): output dependence	
Read after Read (RAR): input dependence	not a dependence

Data Dependence Classification

Examples

TRUE / FLOW / RAW

S1: X = ...

S2: ... = X

value read at S2

depends on

value written at S1

ANTI / WAR

S1: ... = X

S2: X = ...

read at S1

must occur before

write at S2

OUTPUT / WAW

S1: X = ...

S2: X = ...

write at S2 must

occur after write

at S1 if X is read

in a later statement

INPUT / RAR

S1: ... = X

S2: ... = X

read at S1

must occur before

write at S2

(just for
completeness; RAR
is only relevant for
certain
optimizations)

Automatic Parallelization

- ✓ The Scope of Auto-Parallelization
- ✓ Loop Parallelization 101
- ◆ Most Influential Dependence Removing Transformations
- ◆ User's Role in "Automatic" Parallelization
- ◆ Performance of Parallelizing Compilers

Dependence-Removing Program Transformations I: Data Privatization

```
C$OMP PARALLEL DO
C$OMP+ PRIVATE(work)
DO i = 1, n
    work[1:n] = ...
    .
    .
    .
    ... = work[1:n]
ENDDO
```

Dependency:
Elements of `work`
read in iteration i'
were also written
in iteration $i' - 1$.

Each processor is given a separate version of the private data, so there is no sharing conflict

Privatization

loop-carried
anti dependence

```
DO i=1,n
  t = A(i)+B(i)
  C(i) = t + t**2
ENDDO
```

scalar privatization

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(t)
DO i=1,n
  t = A(i)+B(i)
  C(i) = t + t**2
ENDDO
```

```
DO j=1,n
  t(1:m) = A(j,1:m)+B(j)
  C(j,1:m) = t(1:m) + t(1:m)**2
ENDDO
```

array privatization

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(t)
DO j=1,n
  t(1:m) = A(j,1:m)+B(j)
  C(j,1:m) = t(1:m) + t(1:m)**2
ENDDO
```

Array Privatization

```
k = 5
DO j=1,n
  t(1:10) = A(j,1:10)+B(j)
  C(j,iv) = t(k)
  t(11:m) = A(j,11:m)+B(j)
  C(j,1:m) = t(1:m)
ENDDO
```

**More complicated patterns for
Array Privatization**

```
DO j=1,n
  IF (cond(j))
    t(1:m) = A(j,1:m)+B(j)
    C(j,1:m) = t(1:m) + t(1:m)**2
  ENDIF
  D(j,1) = t(1)
ENDDO
```


Dependence-Removing Program Transformations II: Reduction Recognition

```
C$OMP PARALLEL DO
C$OMP+ REDUCTION (+:sum)
DO i = 1, n
  ...
  sum = sum + A[i]
  ...
ENDDO
```

Dependency:
Value of sum
written in iteration
 $i' - 1$ is read in
iteration i' .

Each processor will accumulate partial sums, followed by a combination of these parts at the end of the loop.

Rules for Reductions

- reduction statements in a loop have the form $X = X \otimes \text{expr}$,
 - where X is either scalar or an array expression ($a[\text{sub}]$, where sub must be the same on the LHS and the RHS),
 - \otimes is a reduction operation, such as $+$, $*$, \min , \max
- X must not be used in any non-reduction statement in this loop (however, there may be multiple reduction statements for X)

Reduction Parallelization

```
DO j=1,n  
  sum = sum + a(j)  
  ...  
ENDDO
```

```
DO PARALLEL j=1,n  
  PRIVATE s=0  
    s = s + a(j)  
    ...  
  POSTAMBLE  
    ATOMIC:  
      sum=sum+s  
ENDDO
```

Preamble

Postamble

```
DO PARALLEL j=1,n  
  ATOMIC:  
    sum = sum + a(j)  
  ...  
ENDDO
```

Preamble and Postamble are executed exactly once by all participating threads.

Reduction Parallelization II

Scalar Reductions

```
DO i=1,n  
    sum = sum + A(i)  
ENDDO
```

Remember, OpenMP has a reduction clause;
only reduction recognition is needed:

```
!$OMP PARALLEL DO  
!$OMP+REDUCTION(+:sum)  
DO i=1,n  
    sum = sum + A(i)  
ENDDO
```

```
!$OMP PARALLEL PRIVATE(s)  
s=0  
!$OMP DO  
DO i=1,n  
    s=s+A(i)  
ENDDO  
!$OMP ATOMIC  
sum = sum+s  
!$OMP END PARALLEL
```

*Privatized
reduction
implementation*

```
DO i=1,num_proc  
    s(i)=0  
ENDDO  
!$OMP PARALLEL DO  
DO i=1,n  
    s(my_proc)=s(my_proc)+A(i)  
ENDDO  
DO i=1,num_proc  
    sum=sum+s(i)  
ENDDO
```

*Expanded
reduction
implementation*

Reduction Parallelization III

Array Reductions (a.k.a. irregular or histogram reductions)

```
DIMENSION sum(m)
DO i=1,n
  sum(expr) = sum(expr) + A(i)
ENDDO
```

```
DIMENSION sum(m),s(m)
!$OMP PARALLEL PRIVATE(s)
s(1:m)=0
!$OMP DO
DO i=1,n
  s(expr)=s(expr)+A(i)
ENDDO
!$OMP ATOMIC
sum(1:m) = sum(1:m)+s(1:m)
!$OMP END PARALLEL
```

```
DIMENSION sum(m),s(m,#proc)
!$OMP PARALLEL DO
DO i=1,m
DO j=1,#proc
  s(i,j)=0
ENDDO
ENDDO
!$OMP PARALLEL DO
DO i=1,n
  s(expr,my_proc)=s(expr,my_proc)+A(i)
ENDDO
!$OMP PARALLEL DO
DO i=1,m
DO j=1,#proc
  sum(i)=sum(i)+s(i,j)
ENDDO
ENDDO
```

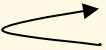
Note, OpenMP 1.0 does not support such array reductions

29

Performance Considerations for Reduction Parallelization

- ◆ Parallelized reductions execute substantially more code than their serial versions \Rightarrow overhead if the reduction (n) is small.
- ◆ In many cases (for large reductions) initialization and sum-up are insignificant.
- ◆ False sharing can occur, especially in expanded reductions, if multiple processors use adjacent array elements of the temporary reduction array (s).
- ◆ Expanded reductions exhibit more parallelism in the sum-up operation compared to privatized reductions.
- ◆ Potential overhead in initialization, sum-up, and memory used for large, sparse array reductions \Rightarrow compression schemes can become useful.

Dependence-Removing Program Transformations III: Induction Variable Substitution

loop-carried
flow dependence 

```
ind = k
DO i=1,n
  ind = ind + 2
  A(ind) = B(i)
ENDDO
```

→ Parallel DO i=1,n
A(k+2*i) = B(i)
ENDDO

This is the simple case of an induction variable

Generalized Induction Variables

```
ind=k
DO j=1,n
  ind = ind + j
  A(ind) = B(j)
ENDDO
```

→

```
Parallel DO j=1,n
  A(k+(j**2+j)/2) = B(j)
ENDDO
```

```
DO i=1,n
  ind1 = ind1 + 1
  ind2 = ind2 + ind1
  A(ind2) = B(i)
ENDDO
```

```
DO i=1,n
  DO j=1,i
    ind = ind + 1
    A(ind) = B(i)
  ENDDO
ENDDO
```


Rules for Induction Variables

- ◆ induction statements in a loop nest have the form
 $iv = iv + expr$ or $iv = iv * expr$,
 where iv is a scalar integer
- ◆ $expr$ must be loop-invariant or another induction variable (there must not be cyclic relationships among IVs)
- ◆ iv must not be assigned in a non-induction statement

User's Role: Choice of Compiler Options

Examples of parallelizing compiler options (typically there is a large set of options)

- optimization levels

Will primarily increase compilation time

- ▼ optimize : **simple analysis, advanced analysis, alias analysis, data-dependence analysis, locality enhancement, array privatization/reduction**
- ▼ aggressive: **data padding, data layout adjustment**

- subroutine inline expansion

Makes up for lack of interprocedural analysis

- ▼ **inline all, specific routines, how to deal with libraries**

- try specific optimizations

May enhance OR degrade performance

- ▼ **e.g., recurrence and reduction recognition, loop fusion, tiling**

More About Compiler Options

- Limits on amount of optimization:
 - ▼ e.g., size of optimization data structures, number of optimization variants tried
- Make certain assumptions:
 - ▼ e.g., array bounds are not violated, arrays are not aliased
- Machine parameters:
 - ▼ e.g., cache size, line size, mapping
- Listing control

Compiler options can be a substitute for advanced compiler strategies.
If the compiler has limited information, the user can help out.

Compiler options are very important for the user to know.
Setting good compiler options can make a *big performance difference*.

User Tuning: Inspecting the Translated Program

- ◆ Source-to-source restructurers:
 - Transformed source code is the actual output
 - Example: Cetus
 - ⇒ The output can be a starting point for code tuning
- ◆ Code-generating compilers:
 - Some have an option for viewing the translated (parallel) code
 - Example: SGI f77 -apo -mplist
 - ⇒ You may modify the source code to make it easier for the compiler to detect parallelism

Compiler Listings

The listing gives many useful clues for improving the performance:

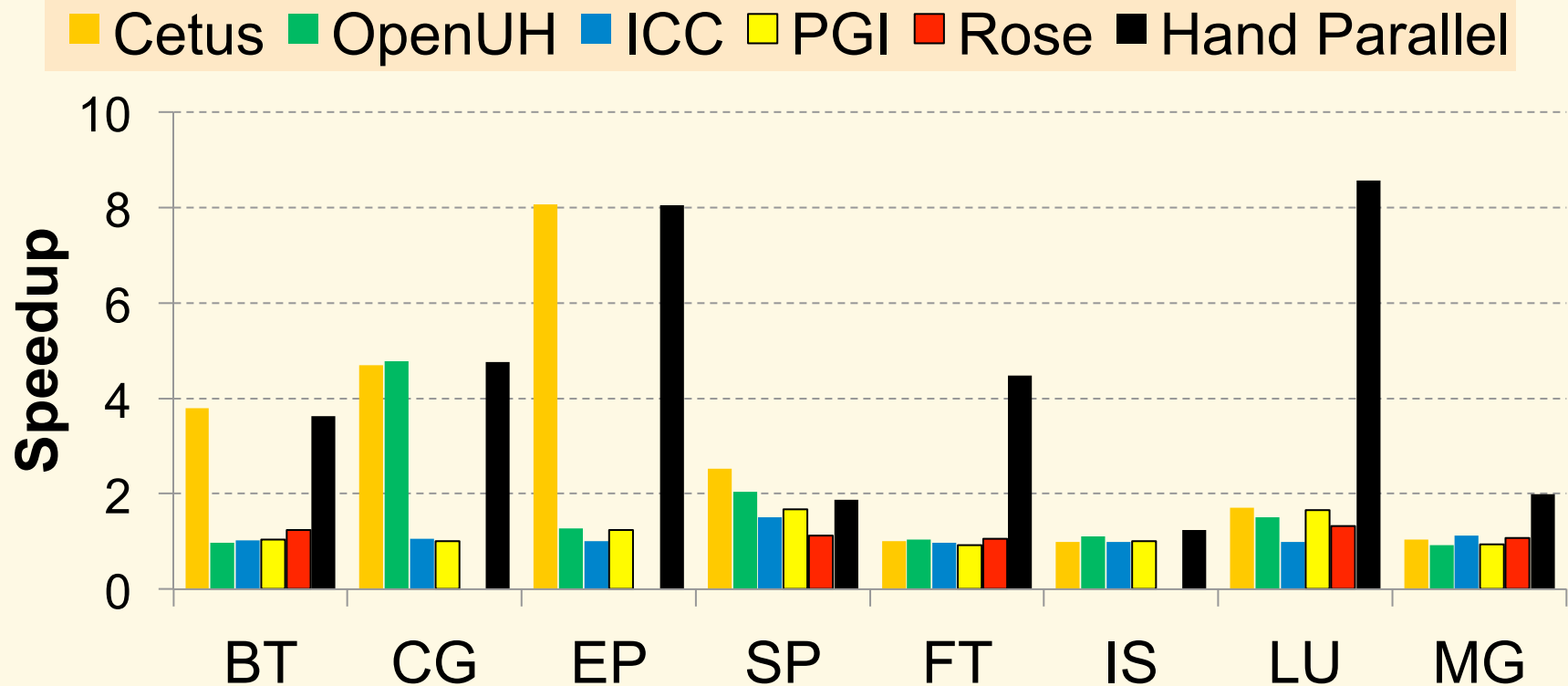
- Loop optimization
 - Reports about data dependencies
 - Explanations about loop dependencies
 - The annotated, transformed code
 - Calling tree
 - Performance statistics
- Loop nest summary:
CLENMO_do#1: loop is parallel
CLENMO_do#1#1: loop is serial, because it contains I/O statements, and the following variables (*may*) have loop-carried dependences: VEC[]

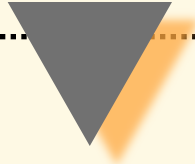
The type of reports to be included in the listing can be set through compiler options.

Automatic Parallelization

- ✓ The Scope of Auto-Parallelization
- ✓ Loop Parallelization 101
- ✓ Most Influential Dependence Removing Transformations
- ✓ User's Role in "Automatic" Parallelization
- ◆ Performance of Parallelizing Compilers

Performance of Parallelizing Compilers





Tuning Automatically-Parallelized Programs

Why Would We Want to Tune Automatically-parallelized Code?

Because

- ◆ compiler techniques are limited
 - E.g., array reductions are parallelized by only few compilers
- ◆ compilers may have insufficient information
 - E.g.,
 - loop iteration range may be input data
 - variables are defined in other subroutines (no interprocedural analysis)
- ◆ Tuning the compiler-parallelized program is generally easier than hand parallelizing a sequential program.

Methods for Tuning Automatically-Parallelized Programs

- ◆ 1. Tuning compiler options
 - Parallelizers have many more options than standard compilers
- ◆ 2. Changing the source program
 - so that the parallelizer can recognize more opportunities for optimizations
- ◆ 3. Manually improving the transformed code
 - This task is similar to explicit parallel programming.
 - The parallelizer must be a source-to-source restructurer.

Tuning Parallelizer Options

- ◆ Tuning compiler options may improve performance by 100% or more.
- ◆ Examples:
 - compile for specific machine architecture
 - enable/disable recurrence recognition
 - padding data structures
 - enable/disable tiling
 - set parallelization threshold
 - set degree of inlining
 - strict language standard interpretation

Changing the Source Program

- ◆ **Inserting directives, such as**
 - explicitly parallel loops (e.g., OpenMP syntax)
 - properties of data structures (e.g., permutation array)
 - assert independence
- ◆ **Modifying the source code.**

Examples:

 - assigning explicit values to variables
 - removing pointers and obscure code
 - removing debug output statements

Manually Improving the Transformed Code

- ◆ The basic method:
 - inspect the most time-consuming loops. If they are not parallelized, find out why; then transform them by hand.
- ◆ Remember (very important):
 - The compiler gives hints in its listing, which may tell you where to focus attention. E.g., which variables have data dependences.

Exercise 1:

A multi-threaded “Hello world” program

- ◆ Write a multithreaded program where each thread prints “hello world”.

```
#include "omp.h"
void main()
{
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
}
```

Sample Output:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

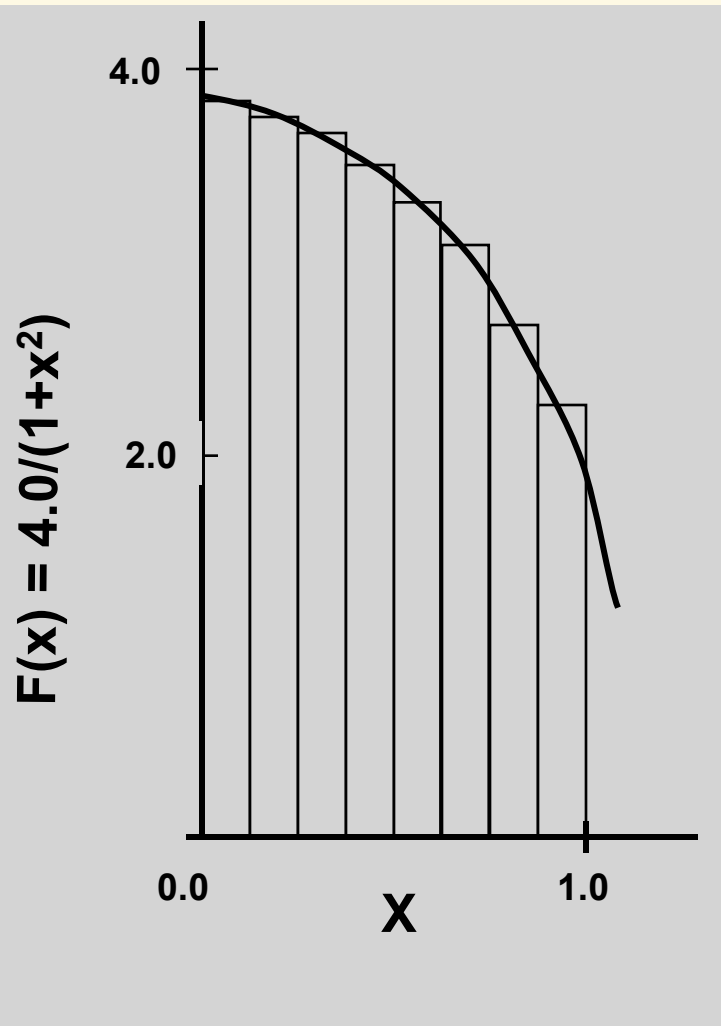
Exercise 2:

A multi-threaded “pi” program

- ◆ On the following slide, you’ll see a sequential program that uses numerical integration to compute an estimate of PI.
- ◆ Parallelize this program using OpenMP. There are several options (do them all if you have time):
 - ▼ Do it as an SPMD program using a parallel region only.
 - ▼ Do it with a work sharing construct.
- ◆ Remember, you’ll need to make sure multiple threads don’t overwrite each other’s variables.

Our running Example: The PI program

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ in the middle of interval i .

PI Program: The sequential program

```
static long num_steps = 100000;
double step;
void main ()
{
    int i;    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

PI Program

The OpenMP parallel version

```
#include <omp.h>
static long num_steps = 100000;
double step;
{
    int i;    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

**OpenMP adds
2 lines of code**

OpenMP PI Program:

Parallelized without a reduction clause

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int i;    double x, pi, sum[NUM_THREADS] = {0.0};
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    double x;    int i, id;
    id = omp_get_thread_num();
#pragma omp for
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
}
```

OpenMP PI Program:

Without the use of a worksharing (omp for) construct => SPMD program

```
#include <omp.h>
static long num_steps = 100000;      double step;
void main ()
{
    int i;      double x, pi, sum[NUM_THREADS] = {0};
    step = 1.0/(double) num_steps;

    #pragma omp parallel
    {
        double x;   int id, i;
        id = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        for (i=id;i< num_steps; i=i+nthreads){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }

    for(i=0, pi=0.0;i<NUM_THREADS;i++) pi += sum[i] * step;
}
```

SPMD Program:

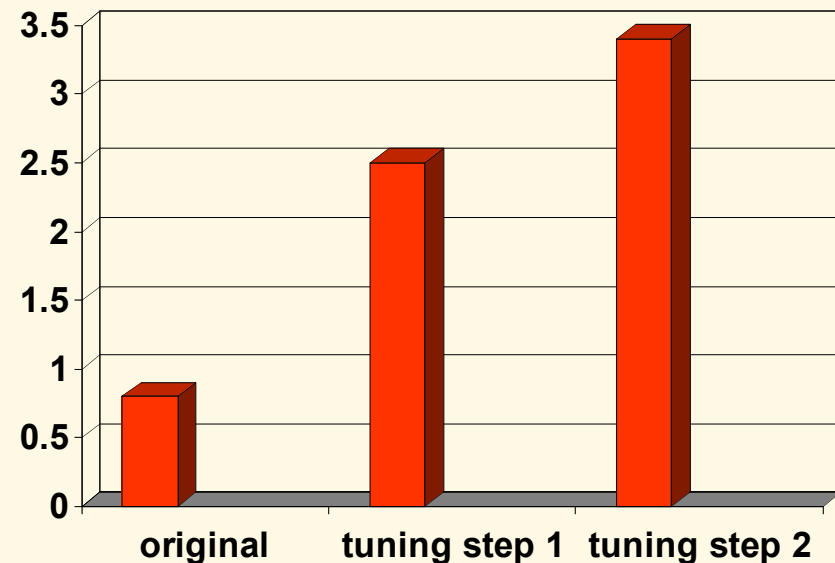
Each thread runs the same code; the thread ID selects any thread-specific behavior.

Performance Tuning

Example 1: MDG

- ◆ MDG: A Fortran code of the “Perfect Benchmarks”.
- ◆ Advanced autoparallelizers may recognize the parallelism in this code.

Performance on a
4-core machine



MDG: Tuning Steps

Step 1: Parallelize the most time-consuming loop. It consumes 95% of the serial execution time.

The transformations it takes to parallelize this loop are:

- array privatization
- reduction parallelization

Step 2: Balancing the iteration space of this loop.

- Loop nest is “triangular”. Default block partitioning would create unbalanced assignment of iterations to processors.

MDG

Structure of the most time-consuming loop in MDG:

```
c1 = x(1)>0
c2 = x(1:10)>0

DO i=1,n
  DO j=i,n
    IF (c1) THEN rl(1:100) = ...
    ...
    IF (c2) THEN ... = rl(1:100)
    sum(j) = sum(j) + ...
  ENDDO
ENDDO
```

Original

```
c1 = x(1)>0
c2 = x(1:10)>0
```

Parallel

```
Allocate(xsum(n,1:#proc))
```

```
C$OMP PARALLEL DO
C$OMP+ PRIVATE (l,j,rl,id)
C$OMP+ SCHEDULE (STATIC,1)
```

```
DO i=1,n
  id = omp_get_thread_num()
```

```
  DO j=i,n
```

```
    IF (c1) THEN rl(1:100) = ...
```

```
    ...
```

```
    IF (c2) THEN ... = rl(1:100)
```

```
    xsum(j,id) = xsum(j,id) + ...
```

```
  ENDDO
```

```
ENDDO
```

```
C$OMP PARALLEL DO
```

```
DO i=1,n
```

```
  sum(i)=sum(i)+xsum(j,1:#proc)
```

```
ENDDO
```

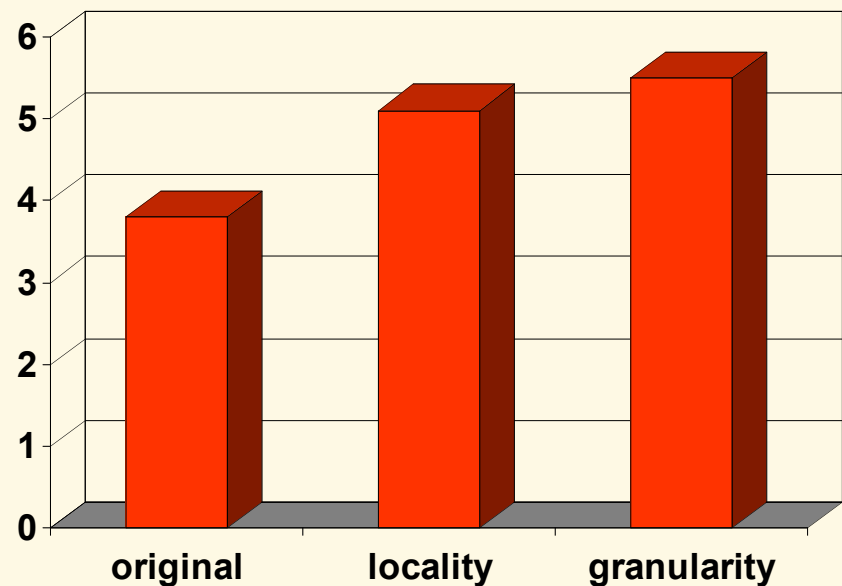
55

Performance Tuning

Example 2: ARC2D

ARC2D: A Fortran code of the “Perfect Benchmarks”.

ARC2D is parallelized very well by available compilers. However, the mapping of the code to the machine could be improved.



ARC2D: Tuning Steps

- ◆ Step 1:
Loop interchanging increases cache locality through stride-1 references
- ◆ Step 2:
Move parallel loops to outer positions
- ◆ Step 3:
Move synchronization points outward
- ◆ Step 4:
Coalesce loops

ARC2D

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(R1,R2,K,J)
→ DO j = jlow, jup
→ DO k = 2, kmax-1
    r1 = prss(jminu(j), k) + prss(jplus(j), k) + (-2.)*prss(j, k)
    r2 = prss(jminu(j), k) + prss(jplus(j), k) + 2.*prss(j, k)
    coef(j, k) = ABS(r1/r2)
  ENDDO
ENDDO
!$OMP END PARALLEL
```

Loop interchanging increases (spatial) cache locality

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(R1,R2,K,J)
→ DO k = 2, kmax-1
→ DO j = jlow, jup
    r1 = prss(jminu(j), k) + prss(jplus(j), k) + (-2.)*prss(j, k)
    r2 = prss(jminu(j), k) + prss(jplus(j), k) + 2.*prss(j, k)
    coef(j, k) = ABS(r1/r2)
  ENDDO
ENDDO
!$OMP END PARALLEL
```

ARC2D

```
!$OMP PARALLEL
!$OMP+PRIVATE(LDI,LD2,LD1,J,LD,K)
  DO k = 2+2, ku-2, 1
!$OMP DO
  DO j = jl, ju
    ld2 = a(j, k)
    ld1 = b(j, k)+(-x(j, k-2))*ld2
    ld = c(j, k)+(-x(j, k-1))*ld1+(-y(j, k-1))*ld2
    ldi = 1./ld
    f(j, k, 1) = ldi*(f(j, k, 1)+(-f(j, k-2, 1))*ld2+(-f(j, k-1, 1))*ld1)
    f(j, k, 2) = ldi*(f(j, k, 2)+(-f(j, k-2, 2))*ld2+(-f(j, k-2, 2))*ld1)
    x(j, k) = ldi*(d(j, k)+(-y(j, k-1))*ld1)
    y(j, k) = e(j, k)*ldi
  ENDDO
!$OMP END DO NOWAIT
  ENDDO
!$OMP END PARALLEL
```

Increasing
parallel loop
granularity
through
NOWAIT clause

Note that

- the k-loop is executed in sequential order and
- all iterations that have the same value j_x are executed by the same thread in all iterations of the k-loop

Related Technique: Loop Blocking

```

DO j=1,m
  DO i=1,n
    B(i,j)=A(i,j)+A(i,j-1)
  ENDDO
ENDDO
  
```

→

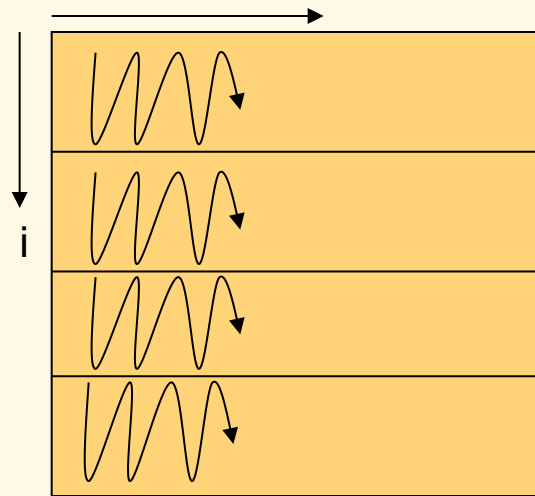
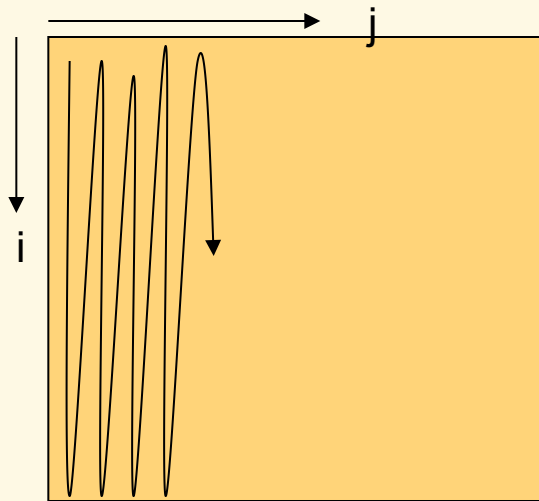
```

DO i1=1,n,block
  DO j=1,m
    DO i=i1,min(i1+block-1,n)
      B(i,j)=A(i,j)+A(i,j-1)
    ENDDO
  ENDDO
ENDDO
  
```

The same effect is achieved like this:

```

!$OMP PARALLEL
DO j=1,m
!$OMP DO SCHEDULE(STATIC,block)
  DO i=1,n
    B(i,j)=A(i,j)+A(i,j-1)
  ENDDO
!$OMP END DO NOWAIT
ENDDO
!$OMP END PARALLEL
  
```



Loop blocking
increases
temporal
locality

Step1: Split inner loop in two (a.k.a. loop “stripmining”)
Step2: interchange outer two loops

ARC2D

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(n, k,j)
  DO n = 1, 4
  DO k = 2, kmax-1
  DO j = jlow, jup
    q(j, k, n) = q(j, k, n)+s(j, k, n)
    s(j, k, n) = s(j, k, n)*phic
  ENDDO
  ENDDO
  ENDDO
!$OMP END PARALLEL
```

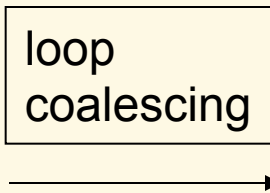
```
!$OMP PARALLEL DO
!$OMP+PRIVATE(nk,n,k,j)
  DO nk = 0,4*(kmax-2)-1
  n = nk/(kmax-2) + 1
  k = MOD(nk,kmax-2)+2
  DO j = jlow, jup
    q(j, k, n) = q(j, k, n)+s(j, k, n)
    s(j, k, n) = s(j, k, n)*phic
  ENDDO
  ENDDO
!$OMP END PARALLEL
```

Increasing parallel loop granularity
though loop coalescing (a.k.a. loop collapsing)

Underlying Technique: Loop Coalescing/Collapsing

```
PARALLEL DO i=1,n  
  DO j=1,m  
    A(i,j) = B(i,j)  
  ENDDO  
ENDDO
```

loop
coalescing



```
PARALLEL DO ij=1,n*m  
  i = 1 + (ij-1) DIV m  
  j = 1 + (ij-1) MOD m  
  A(i,j) = B(i,j)  
ENDDO
```

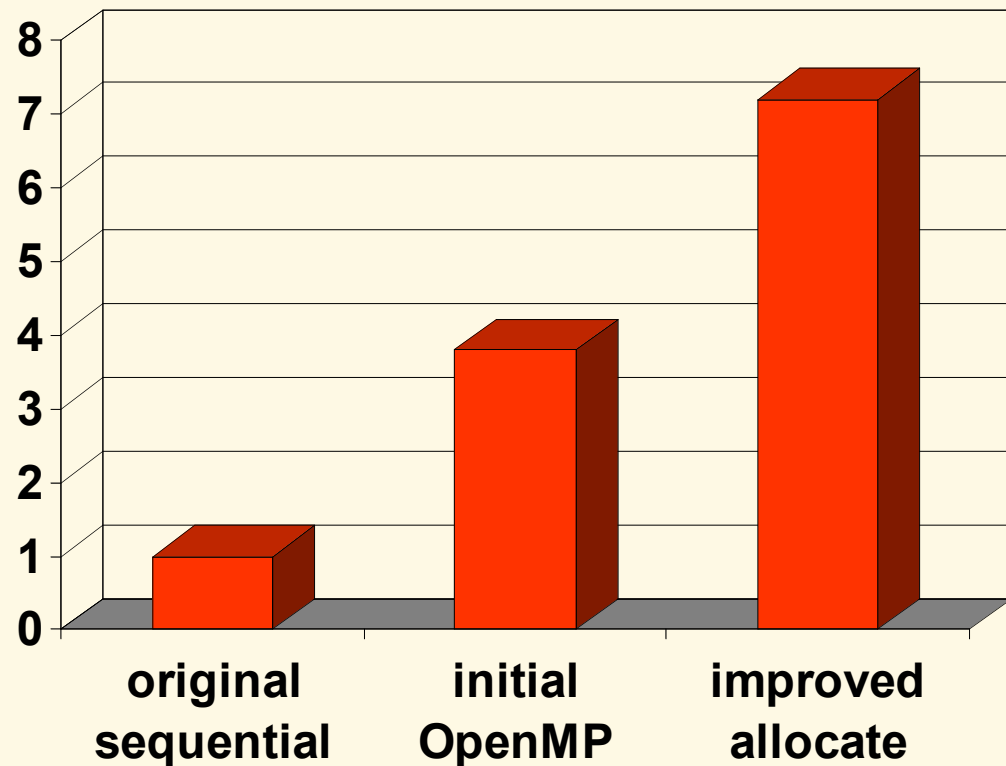
Transformation can be beneficial if

- n is small, unknown, or variable
- the loop body is large
- computation is irregular

Performance Tuning Example 3: EQUAKE

EQUAKE: A C code of the SPEC OMP
2001 benchmarks.

EQUAKE is hand-parallelized with relatively few code modifications. It achieves excellent speedup.



EQUAKE: Tuning Steps

◆ Step1:

Parallelizing the four most time-consuming loops

- ▼ inserted OpenMP pragmas for parallel loops and private data
- ▼ array reduction transformation

◆ Step2:

A change in memory allocation

EQUAKE

subroutine
smvp

```
/* malloc w1[numthreads][...] */  
#pragma omp parallel for  
  for (j = 0; j < numthreads; j++)  
    for (i = 0; i < nodes; i++) { w1[j][i] = 0.0; ...; }  
  
#pragma omp parallel private(my_cpu_id,exp,...)  
{  
  my_cpu_id = omp_get_thread_num();  
  
#pragma omp for  
  for (i = 0; i < nodes; i++)  
    while (...) {  
      ...  
      exp = loop-local computation;  
      w1[my_cpu_id][...] += exp;  
      ...  
    }  
}  
  
#pragma omp parallel for  
  for (j = 0; j < numthreads; j++) {  
    for (i = 0; i < nodes; i++) { w[i] += w1[j][i]; ...;}
```

Example 4: GAFORT

- Parallel shuffle
- 20,000 locks
- Deadlock
- Parallel random number generation

Different executions
produce different
results =>
*asynchronous
algorithm,
non-deterministic
parallelism*

```
!$OMP PARALLEL PRIVATE(rand, iother, itemp, temp)
int my_cpu_id = 1
!$ my_cpu_id = omp_get_thread_num() + 1
!$OMP DO
DO j=1,npopsiz-1
  CALL ran3(1,rand,my_cpu_id,0)
  iother=j+1+DINT(DBLE(npopsiz-j)*rand)
!$  IF (j < iother) THEN
!$    CALL omp_set_lock(lck(j))
!$    CALL omp_set_lock(lck(iother))
!$  ELSE
!$    CALL omp_set_lock(lck(iother))
!$    CALL omp_set_lock(lck(j))
!$  END IF
  itemp(1:nchome)=iparent(1:nchome,iother)
  iparent(1:nchome,iother)=iparent(1:nchome,j)
  iparent(1:nchome,j)=itemp(1:nchome)
  temp=fitness(iother)
  fitness(iother)=fitness(j)
  fitness(j)=temp
!$  IF (j < iother) THEN
!$    CALL omp_unset_lock(lck(iother))
!$    CALL omp_unset_lock(lck(j))
!$  ELSE
!$    CALL omp_unset_lock(lck(j))
!$    CALL omp_unset_lock(lck(iother))
!$  END IF
END DO
!$OMP END DO
!$OMP END PARALLEL
```

subroutine
shuffle

Atomic Region

This is also referred to as *Transactional Memory*

```
!$OMP PARALLEL PRIVATE(rand, iother, itemp, temp)
int my_cpu_id = 1
!$ my_cpu_id = omp_get_thread_num() + 1
!$OMP DO
DO j=1,npopsiz-1
  CALL ran3(1,rand,my_cpu_id,0)
  iother=j+1+DINT(DBLE(npopsiz-i)*rand)
  !$ IF (j < iother) THEN
  !$   CALL omp_set_lock(lck(i))
  !$   CALL omp_set_lock(lck(iother))
  !$ ELSE
  !$   CALL omp_set_lock(lck(iother))
  !$   CALL omp_set_lock(lck(j))
  !$ END IF
  itemp(1:nchrome)=iparent(1:nchrome,iother)
  iparent(1:nchrome,iother)=iparent(1:nchrome,j)
  iparent(1:nchrome,j)=itemp(1:nchrome)
  temp=fitness(iother)
  fitness(iother)=fitness(j)
  fitness(i)=temp
  !$ IF (j < iother) THEN
  !$   CALL omp_unset_lock(lck(iother))
  !$   CALL omp_unset_lock(lck(j))
  !$ ELSE
  !$   CALL omp_unset_lock(lck(j))
  !$   CALL omp_unset_lock(lck(iother))
  !$ END IF
END DO
!$OMP END DO
!$OMP END PARALLEL
```

Atomic {

Execution by each thread appears indivisible

}

Atomic Region vs. Critical Section

```
AtomicRegion {  
  <statements>  
}
```

```
CriticalSection {  
  <statements>  
}
```

All processors execute
<statements> indivisibly



#threads executing
<statements> in parallel

All, but conflicts
force serialization
or rollback

1

Deterministic execution
order

no

no

Implementation complexity

high

low

Parallel Programming Tools and Methodologies

What Tools Did We Use for Performance Analysis and Tuning?

◆ Compilers

- the starting point for performance tuning was the compiler-parallelized program.
- It reports: parallelized loops, data dependences, call graph.

◆ Subroutine and loop profilers

- focusing attention on the most time-consuming loops is absolutely essential.

◆ Performance “spreadsheets”:

- typically comparing performance differences at the loop level.

Guidelines for Fixing “Performance Bugs”

- ◆ The methodology that worked for us:
 - Use compiler-parallelized code as a starting point
 - Get loop profile and compiler listing
 - Inspect time-consuming loops (largest potential for improvement)

- ▼ Case 1. Check for parallelism where the compiler could not find it
- ▼ Case 2. Improve parallel loops where the speedup is limited

Remember: we are considering a program-level approach to performance tuning, as opposed to an application-level approach.

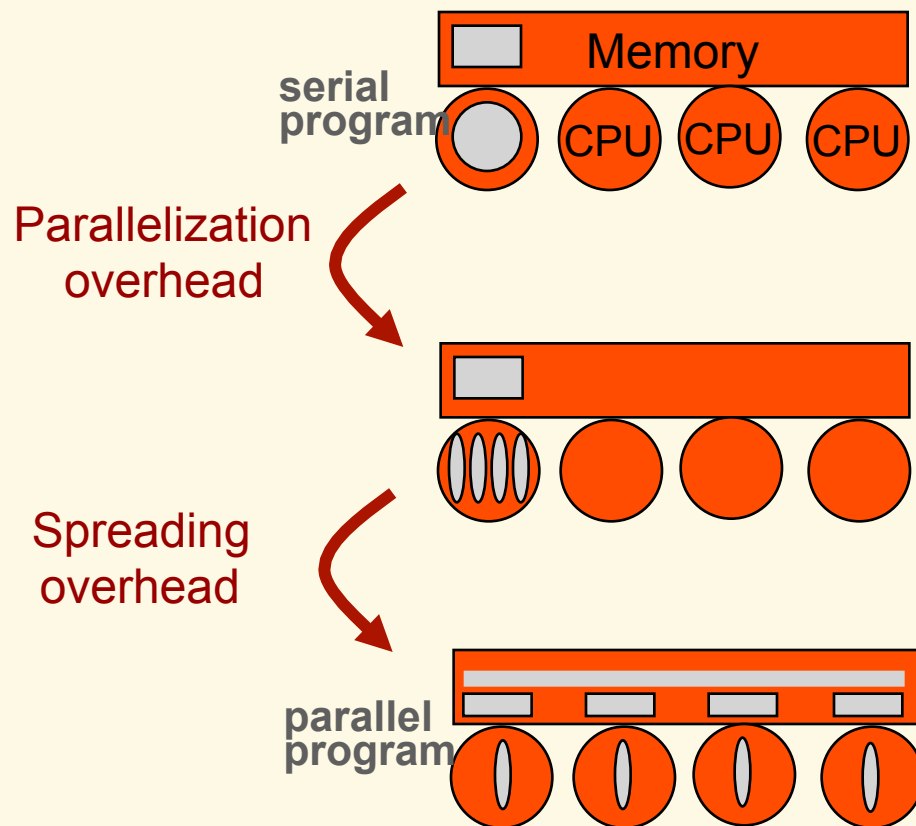
Performance Tuning

Case 1: if the loop is not parallelized automatically, do this:

- ◆ Check for parallelism:
 - read the compiler explanation, if available
 - a variable may be independent even if the compiler detects dependences (compilers are conservative)
 - check if conflicting array is privatizable (e.g., compilers don't perform array privatization well)
 - check if the conflicting variable is a reduction or induction variable
- ◆ If you find parallelism, add OpenMP parallel directives, or make the information explicit for the parallelizer

Performance Tuning

Case 2: if the loop is parallel but does not perform well, consider several optimization factors:



High overheads are caused by:

- parallel startup cost
- small loops
- additional parallel code
- over-optimized inner loops
- less optimization for parallel code

- load imbalance
- synchronized section
- non-stride-1 references
- many shared references
(memory bandwidth)
- low cache affinity
(increased cache misses)

Parallelization Overheads

- ◆ **Parallel startup cost**
 - even with efficient runtime libraries and microtasking schemes, fork/join overheads are in the order of 1-10 μ s. They are machine-specific and may increase with the number of processors.
 - the overhead includes the time to
 - ▼ wakeup helper tasks and communicate data to them (fork), and
 - ▼ the barrier synchronization at the end of the loop (join)
- ◆ **Small loops**
 - loops with small numbers of iterations or small bodies
 - generally, if the average execution time of a loop is less than 0.1 ms, you cannot expect good speedup.
 - ▼ How many statements execute in that time?

Parallelization Overheads continued

- ◆ **Additional parallel code**
 - parallel code may perform more work than the serial code
 - ▼ examples: - initialization and final sum of reductions
 - parallel search may search a larger space than the serial equivalent
- ◆ **Over-optimized inner loops**
 - the compiler may parallelize a subroutine, not knowing that it is called from within a parallel loop
- ◆ **Less optimization for parallel code**
 - compilers are typically more conservative when optimizing parallel code. E.g., Sun's OpenMP compiler uses -O3, although -O5 is available for serial code

Spreading Overheads

◆ Load imbalance

- uneven numbers of iterations
 - ▼ triangular loops
 - ▼ many if statements
 - ▼ non-uniform memory or disk access times
- interruptions
 - ▼ other programs
 - ▼ OS processes

◆ Synchronized section

- even short critical sections can decrease performance substantially
- then there is the cost of calling synchronization primitives

Spreading Overheads continued

◆ Non-stride-1 references

- this is already a problem in serial programs
- issues in parallel programs:
 - ▼ several processors read from the same cache line
=> more cache misses
 - ▼ several processors write to the same cache line
=> false sharing even if the accesses go to different data

◆ Many shared references

- private data can be kept in the local memory (segment)
- bandwidth of shared memory is limited

◆ Low cache affinity

- additional cache misses are incurred if processors access different data in consecutive loops

Profiling Techniques

- ◆ Simple timing on/off calls before/after loops are useful.
- ◆ Optimizations often affect other program sections as well. Be sure to monitor the overall program execution time as well.
- ◆ Both inclusive and exclusive profiles are useful.
- ◆ Be aware that inserted subroutine calls may inhibit certain compiler optimizations.

MDG performance optimization report

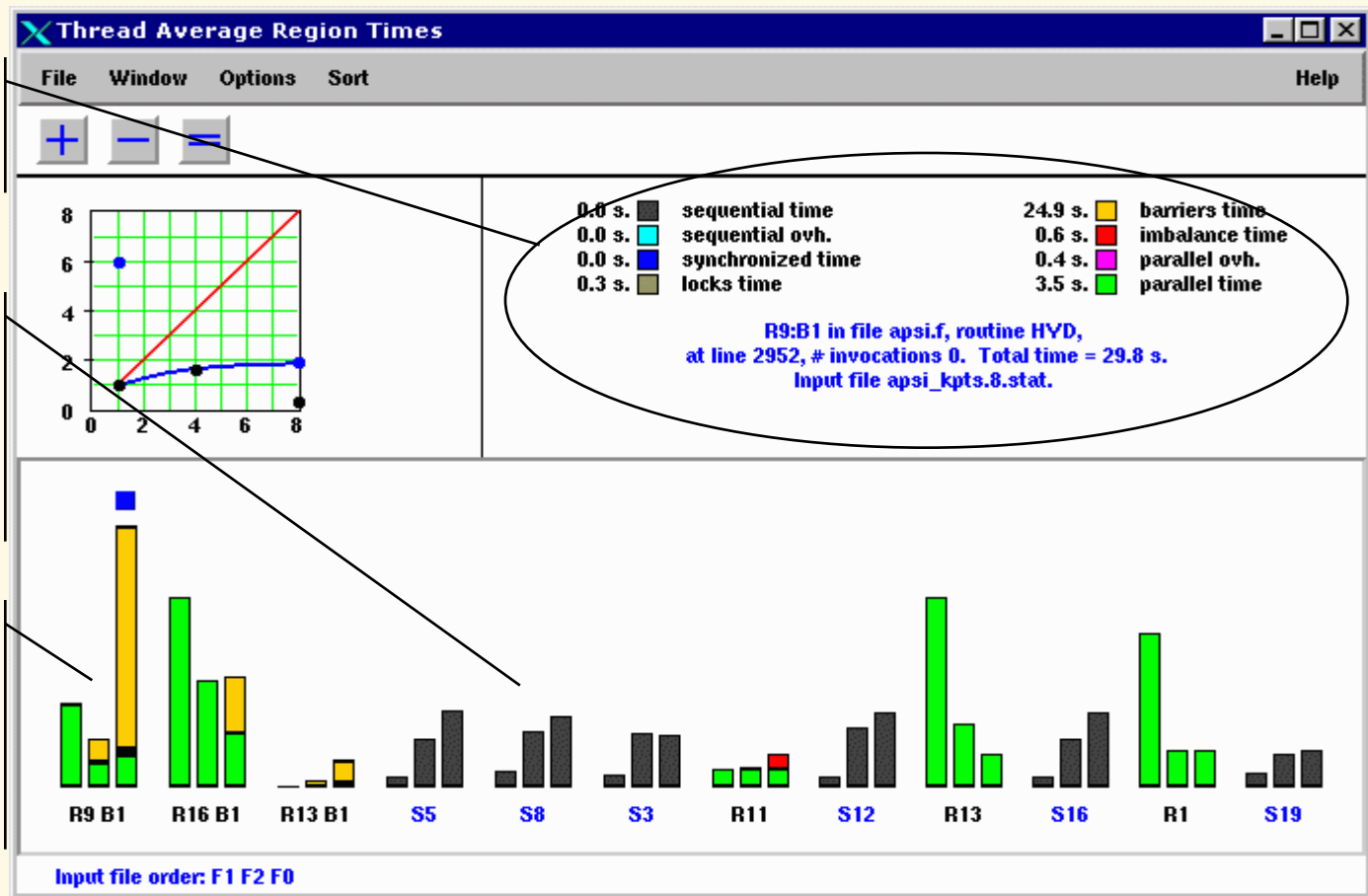
(see handout)

Performance Analysis Tool Examples

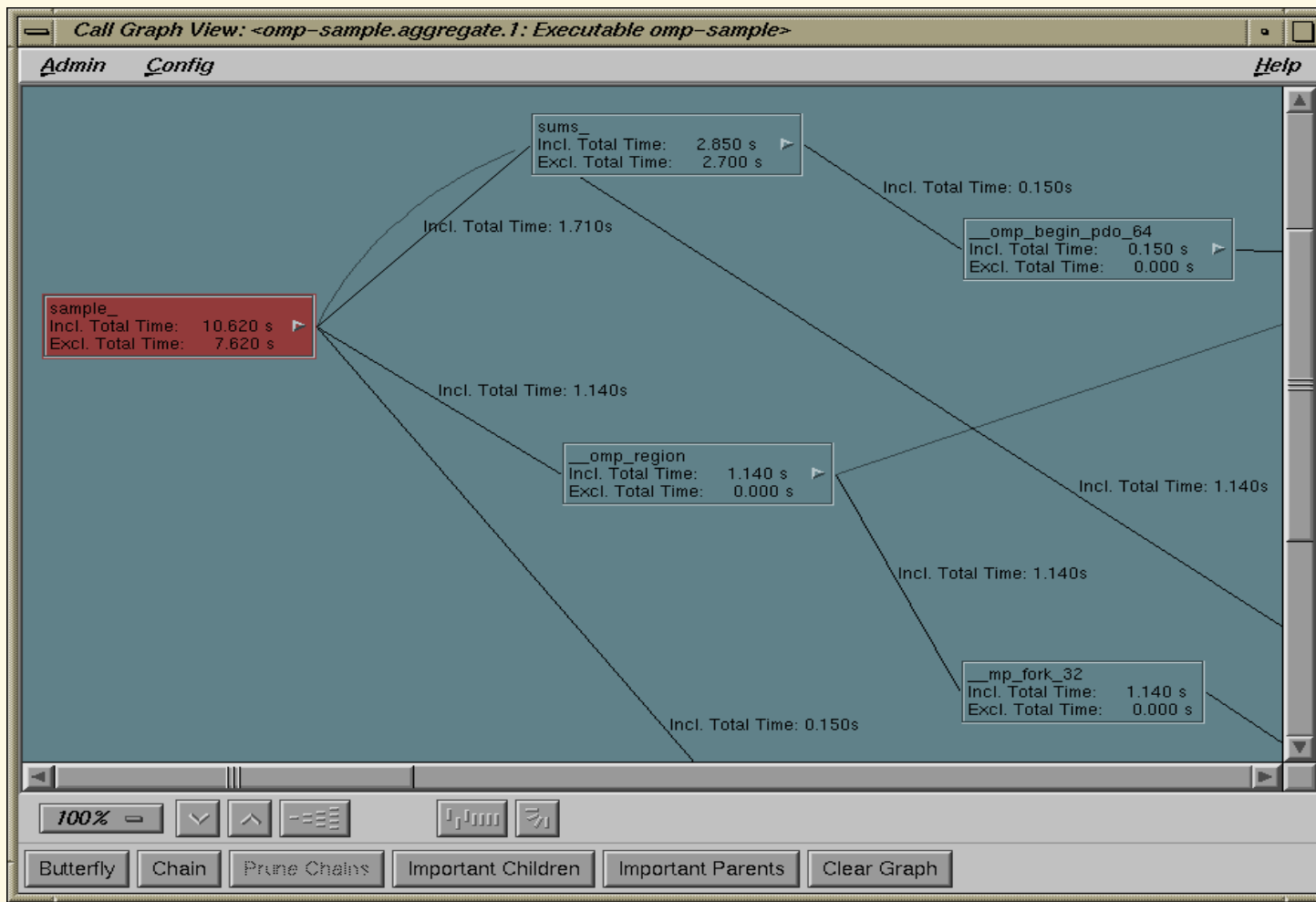
Analyze each Parallel region

Find serial regions that are hurt by parallelism

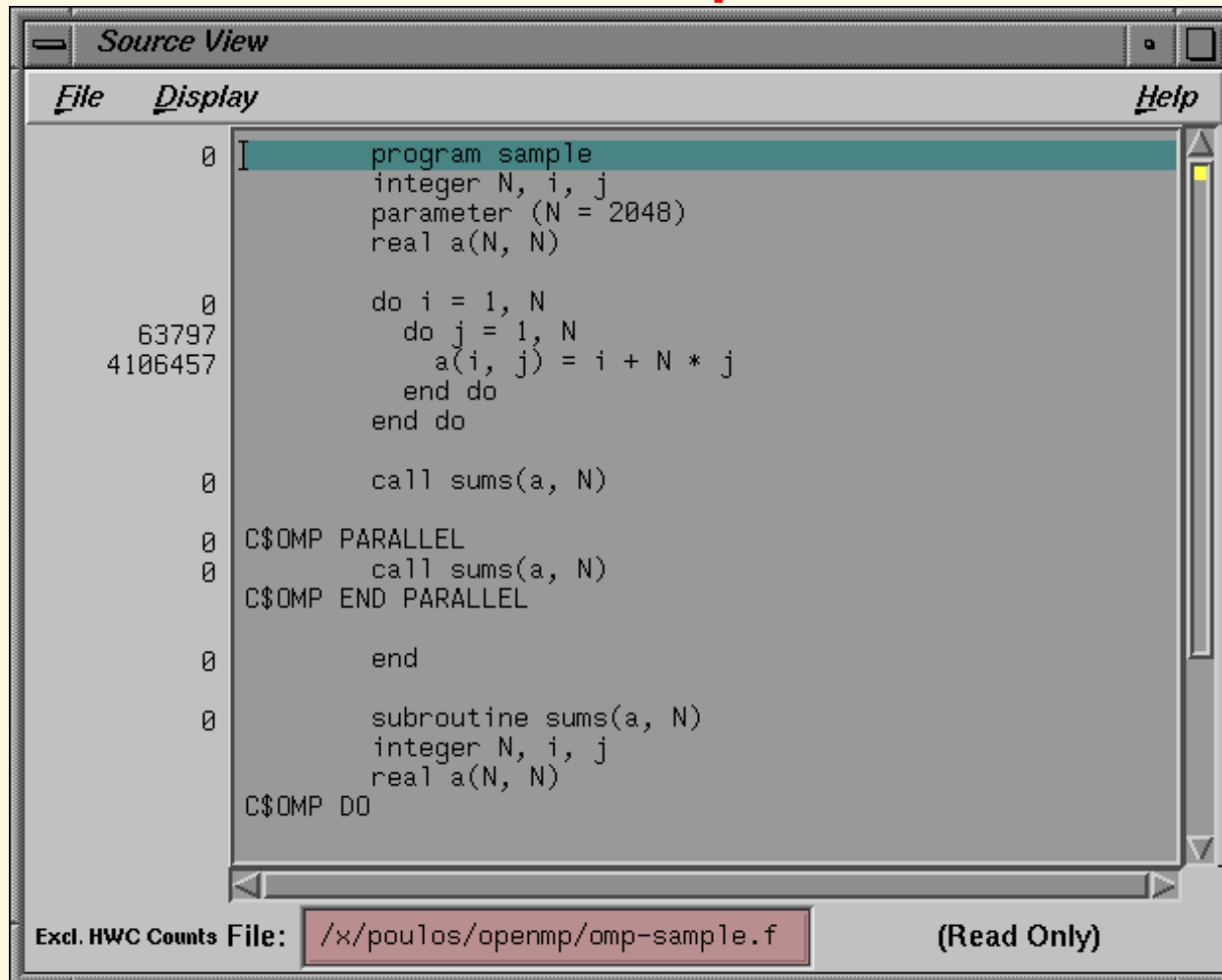
Sort or filter regions to navigate to hotspots



Performance Analysis Tool Examples



Performance Analysis Tool Examples



The screenshot shows a window titled "Source View" with a menu bar containing "File", "Display", and "Help". The main area displays Fortran code with performance metrics on the left. The code includes a program named "sample" with a parameter N=2048, a nested loop for calculating a matrix element, a call to a subroutine "sums", and an OpenMP parallel region. The subroutine "sums" is also shown at the bottom.

```
0 | program sample
0 | integer N, i, j
0 | parameter (N = 2048)
0 | real a(N, N)
0 |
0 | do i = 1, N
63797 |   do j = 1, N
4106457 |     a(i, j) = i + N * j
0 |   end do
0 | end do
0 |
0 | call sums(a, N)
0 |
0 | C$OMP PARALLEL
0 |   call sums(a, N)
0 | C$OMP END PARALLEL
0 |
0 | end
0 |
0 | subroutine sums(a, N)
0 | integer N, i, j
0 | real a(N, N)
0 | C$OMP DO
```

Excl. HWC Counts File: /x/poulos/openmp/omp-sample.f (Read Only)

Performance Analysis Tool Examples

Program Structure View

	Col 4 (TOT)	Col 5 (TOT)	Col 8 (AVG)	Col 9 (90%)	Col 10
ACTFOR_do240	23.292509	6.615926	1.323105	5	3.520
ACTFOR_do500	21.365515	6.411459	1.282292	5	3.332
ACTFOR_do320	0.637071	0.219626	0.052196	100	2.993
RESTAR_do560	0.179432	1.136057	0.560228	2	0.693

Performance Spreadsheet

Introductions to

- ◆ OpenMP

- ◆ MPI

(see separate slides)

PI Program in MPI

```
static long num_steps = 100000;    double step;
void main () {
    int i;    double x, pi, sum =0;
    step = 1.0/(double) num_steps;
    double x;    int id, i;

    MPI_Init(...);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);//rank of each process
    MPI_Comm_size (MPI_COMM_WORLD, &p);//total number of processes

    for (i=id;i< num_steps; i=i+p) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }

    MPI_Allreduce(...sum...pi...);

    MPI_Finalize();
}
```

Introduction to Pthreads

POSIX Standard, IEEE Std 1003.1c-1995

Creating and starting a new thread:

```
int pthread_create (pthread_t *thread,  
                    const pthread_attr_t *attr,  
                    void * (*routine)(void*),  
                    void* arg)
```

- ◆ Returns 0 if successful, or non-zero error code.
- ◆ *thread* points to the ID of the newly created thread.
- ◆ *attr* specifies the thread attributes to be applied to the new thread. NULL for default attributes.
- ◆ *routine* is the name of the function that the thread calls when started. It takes a single parameter (*arg*), a pointer to void.

Introduction to Pthreads

Blocking the calling thread until the specified thread ends:

```
int pthread_join (pthread_t thread,  
                  void ** status);
```

Introduction to Pthreads

More thread management functions

- ◆ **pthread_self** : find out own thread ID
- ◆ **pthread_equal** : test two thread ID for equality
- ◆ **pthread_detach** : set thread to release resources
- ◆ **pthread_exit** : exit thread without exiting the process
- ◆ **pthread_cancel** : terminates another thread

Mutexes

`int pthread_mutex_lock (pthread_mutex_t *mutex);`
locks the mutex. If already locked, blocks caller.

`int pthread_mutex_trylock (pthread_mutex_t *mutex);`
like `mutex_lock`, but no blocking if locked already

`int pthread_mutex_unlock (pthread_mutex_t *mutex);`
wakes 1st thread waiting on *mutex*

Semaphores

- ◆ init semaphore to *value*:

int `sem_init` (int `sem_t` * *sem*, int *p shared*, unsigned int *value*)

- ◆ increments *sem*. **Any** waiting thread will wake up:

int `sem_post` (int `sem_t` * *sem*)

- ◆ decrements *sem*. Blocks if *sem* is 0:

int `sem_wait` (int `sem_t` * *sem*)

Data Sharing in Pthreads

- ◆ Parent and child thread share global variables
- ◆ However, it is not guaranteed that a write to a variable is seen immediately by the other thread
- ◆ Synchronization functions make the global memory state consistent
- ◆ For *volatile* variables, the compiler will generate code that reads from/write to memory at every access.
 - BUT: the architecture may still not guarantee that the memory state is immediately seen by the other thread.

Understanding memory/consistency models is an advanced topic. Whenever possible, use programming constructs that update the memory state implicitly.

- use OpenMP parallel and workshare constructs with implicit barriers
- enclose shared variables with synchronization functions in Pthreads

Translating OpenMP into (P)threads

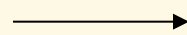
- ◆ Most architectures do not have instructions that support the execution of parallel loops directly
- ◆ A compiler (an OpenMP preprocessor) must translate the source program into a thread-based form.
- ◆ The thread-based form of the program makes calls to a runtime library that supports the OpenMP execution scheme.

Instructions that support direct execution of parallel loops

1. Example architecture: Alliant FX/8 (1980es)

- machine instruction for parallel loop
- HW concurrency bus supports loop scheduling

```
a=0
DO i=1,n
  b(i) = 2
ENDDO
b=3
```

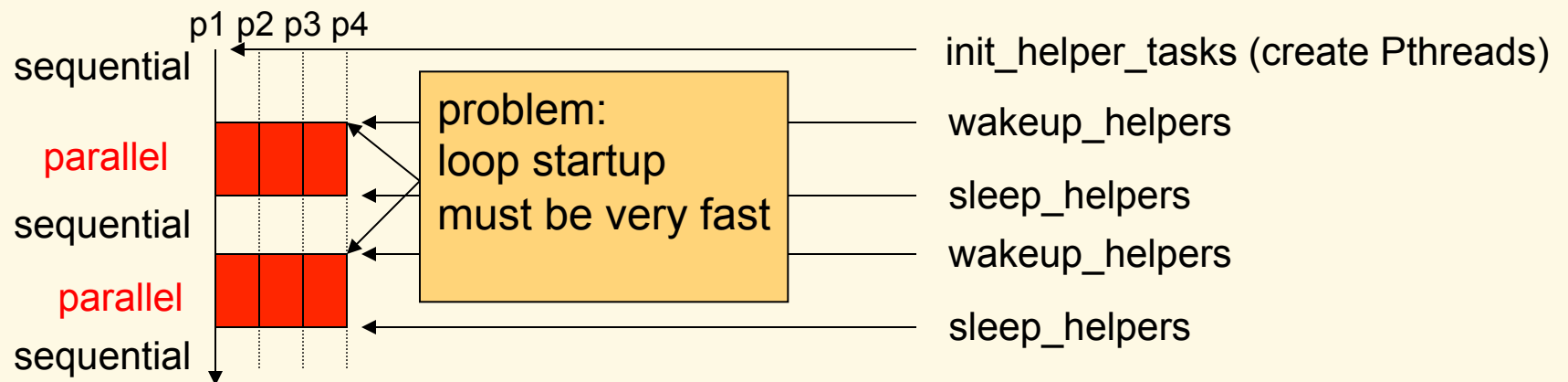


```
store #0,<a>
load <n>,D6
sub 1,D6
load &b,A1
cdoall D6
  store #2,A1(D7.r)
endcdoall
store #3,<b>
```

D7 is reserved
for the loop
variable.
Starts at 0.

Usual Underlying Execution Scheme for Parallel Loops (a.k.a. Microtasking)

2. Microtasking scheme (dates back to early IBM mainframes)



microtask startup: a few μs
pthreads startup: 100 μs or more

Compiler Transformation for the Microtasking Scheme

```
a=0
C$OMP PARALLEL DO
DO i=1,n
  b(i) = 2
ENDDO
b=3
```

```
call init_microtasking() // once at program start
...
a=0
call loop_scheduler(loopsb,i,1,n,b)
b=3
```

```
subroutine loopsb(mytask,lb,ub,b)
DO i=lb,ub
  b(i) = 2
ENDDO
END
```

Loop scheduler:

Master task side

```
loop_scheduler:
partition loop iterations
wakeup
call loopsb(...)
barrier (all flags reset)
return
```

shared data

```
Helper 1:
loopsb
lb,ub
param
flag
```

Helper task side

```
loop:
wait for flag
call loopsb(id,lb,ub,param)
reset flag
```

OpenMP vs MPI

lessons learned from a seismic processing applications

- ◆ MPI and OpenMP can look like opposite ends of some programming method spectrum
- ◆ However, even here *it is true that*
 - it is at least as important *how* you use a model than *what* the model is.
- ◆ Our lesson will show that you can program in MPI style using OpenMP. In particular, the following are myths:
 - MPI exploits outermost parallelism, OpenMP exploits inner parallelism
 - MPI programs have good locality, OpenMP programs do not
 - MPI threads are very different from OpenMP threads

Case Study of a Large-Scale Application

- ◆ Overview of the Application
- ◆ Basic use of MPI and OpenMP
- ◆ Issues Encountered
- ◆ Performance Results

Overview of Seismic

- ◆ Representative of modern seismic processing programs used in the search for oil and gas.
- ◆ 20,000 lines of Fortran. C subroutines interface with the operating system.
- ◆ Available in a serial and a parallel variant.
- ◆ Parallel code is available in a message-passing and an OpenMP form.
- ◆ Is part of the SPEC HPC benchmark suite.
 - SPEC HPC is now retired
- ◆ Includes 4 data sets: small to x-large.

Seismic: Basic Characteristics

- ◆ Program structure:
 - 240 Fortran and 119 C subroutines.
- ◆ Main algorithms:
 - FFT, finite difference solvers
- ◆ Running time of Seismic (@ 500MFlops):
 - small data set: 0.1 hours
 - x-large data set: 48 hours
- ◆ IO requirement:
 - small data set: 110 MB
 - x-large data set: 93 GB

Basic Parallelization Scheme

- ◆ Split into p parallel tasks
(p = number of processors)

MPI

```
Program Seismic
If (rank=0) initialization

call main_subroutine()
```

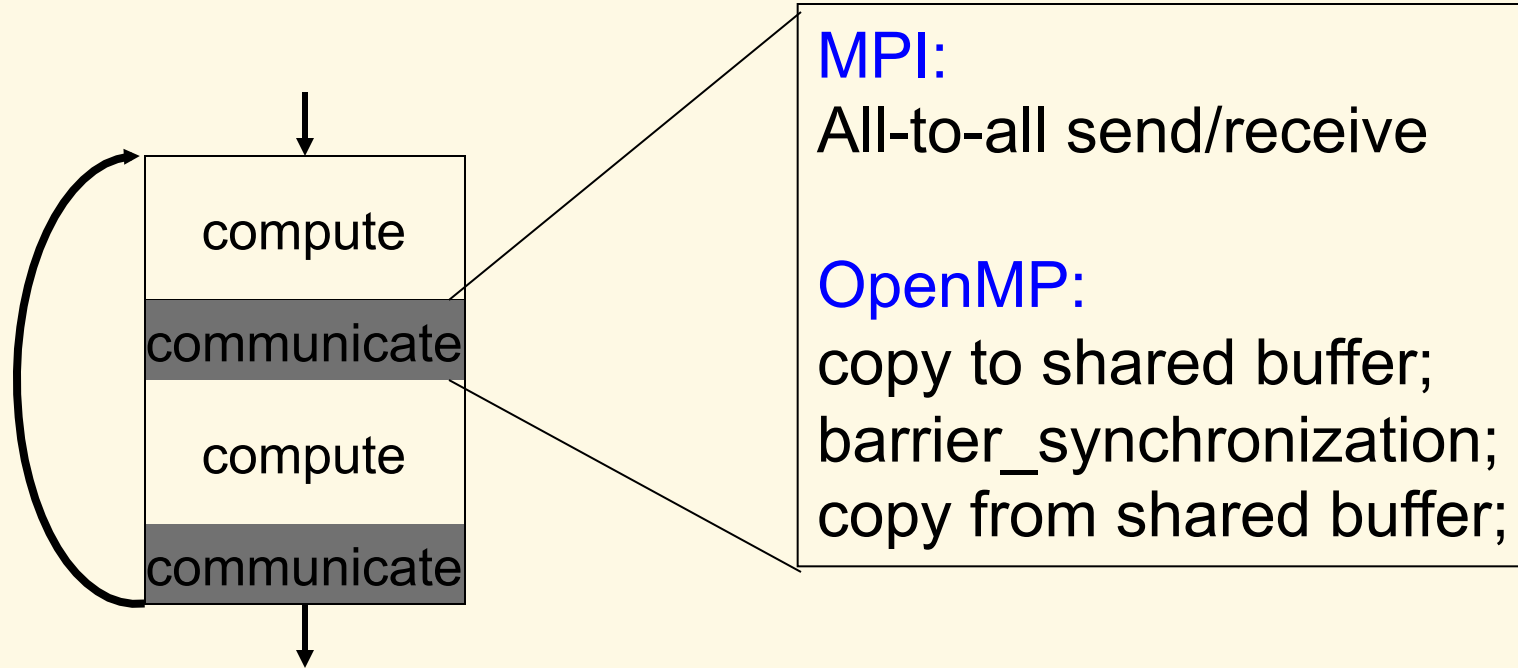
OpenMP

```
Program Seismic
initialization

C$OMP PARALLEL
  call main_subroutine()
C$OMP END PARALLEL
```

→SPMD execution scheme.
Initialization done by master thread only

Synchronization and Communication



Issues:

Mixing Fortran and C

- ◆ Bulk of computation is done in Fortran
- ◆ Utility routines are in C:
 - IO operations
 - data partitioning routines
 - communication/synchronization operations
- ◆ OpenMP-related issues:
 - IF C/OpenMP compiler is not available, data privatization must be done through “expansion”.
 - Mix of Fortran and C is implementation dependent

Data privatization in OpenMP/C

```
#pragma omp thread private (item)  
float item;  
void x(){  
    ... = item;  
}
```

Data expansion in absence of OpenMP/C compiler

```
float item[num_proc];  
void x(){  
    int thread;  
    thread = omp_get_thread_num_();  
    ... = item[thread];  
}
```

Broadcast Common Blocks

```
common /cc/ cdata
common /dd/ ddata
C initialization:
IF (rank=0)
  cdata = ...
  ddata = ...
ENDIF
```

```
C copy common blocks:
call syspcom ()

call main_subroutine()
```

```
common /cc/ cdata
common /dd/ ddata
#pragma OMP threadprivate /cc/,/dd/
c initialization
  cdata = ...
  ddata = ...
```

```
C$OMP PARALLEL
C$OMP+COPYIN(/cc/, /dd/)
  call main_subroutine()
C$END PARALLEL
```

OpenMP issue: At the start of the parallel region it is not yet known which common blocks need to be copied in.

Solution: copy-in all common blocks => overhead

OpenMP Issues: Multithreading IO and malloc

IO routines and memory allocation are called within parallel threads, inside C utility routines.

- ◆ OpenMP requires all standard libraries and intrinsics to be thread-safe. However the implementations are not always compliant.

→ system-dependent solutions need to be found

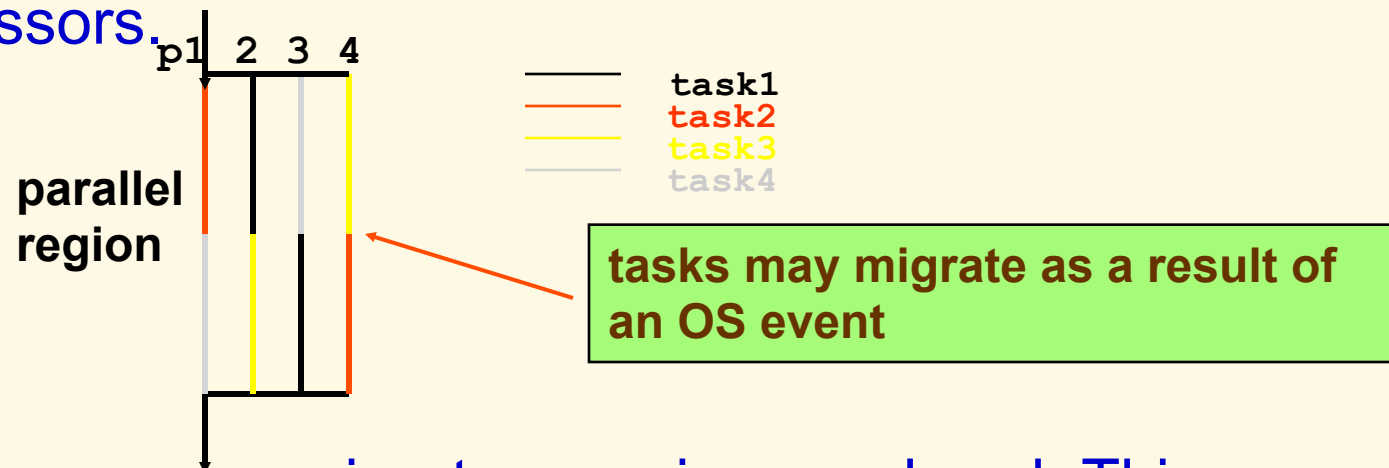
- ◆ The same issue arises if standard C routines are called inside a parallel Fortran region or in non-standard syntax.

Standard C compilers do not know anything about OpenMP and the thread safety requirement.

Note, this is not an issue in MPI

More OpenMP Issues: Processor Affinity

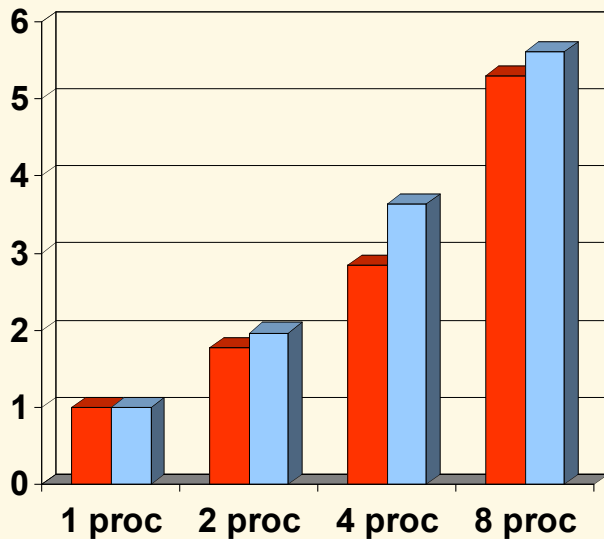
- ◆ OpenMP currently does not specify or provide constructs for controlling the binding of threads to processors.



- ◆ Processors can migrate, causing overhead. This behavior is system-dependent.
System-dependent solutions may be available.

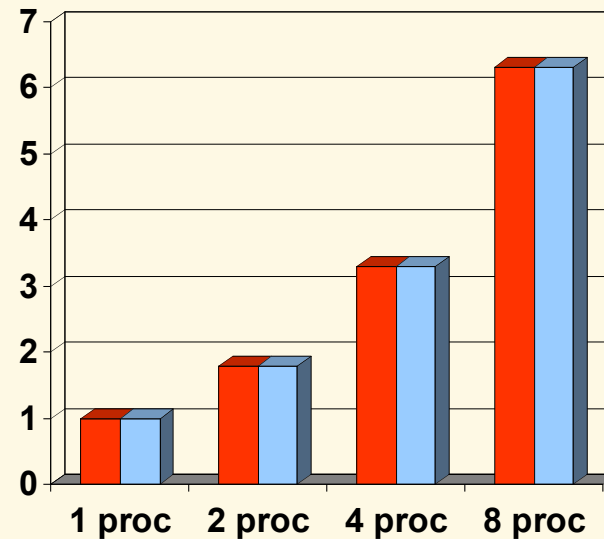
Performance Results

Speedups of Seismic on an 8-processor system



small data set

■ MPI
■ OpenMP



medium data set

Programming Vector Architectures

Vector Instructions

A vector instruction operates on a number of data elements at once.

Example: `vadd va,vb,vc,32`

vector operation of length 32 on vector registers va,vb, and vc

- va,vb,vc can be
 - ▼ Special cpu registers or memory → classical supercomputers
 - ▼ Regular registers, subdivided into shorter partitions (e.g., 64bit register split 8-way) → multi-media extensions
- The operations on the different vector elements can overlap → vector pipelining

Applications of Vector Operations

- ◆ Science/engineering applications are typically regular with large loop iteration counts.

This was ideal for classical supercomputers, which had long vectors (up to 256; vector pipeline startup was costly).

- ◆ Graphics applications can exploit “multi-media” register features and instruction sets.

Basic Vector Transformation

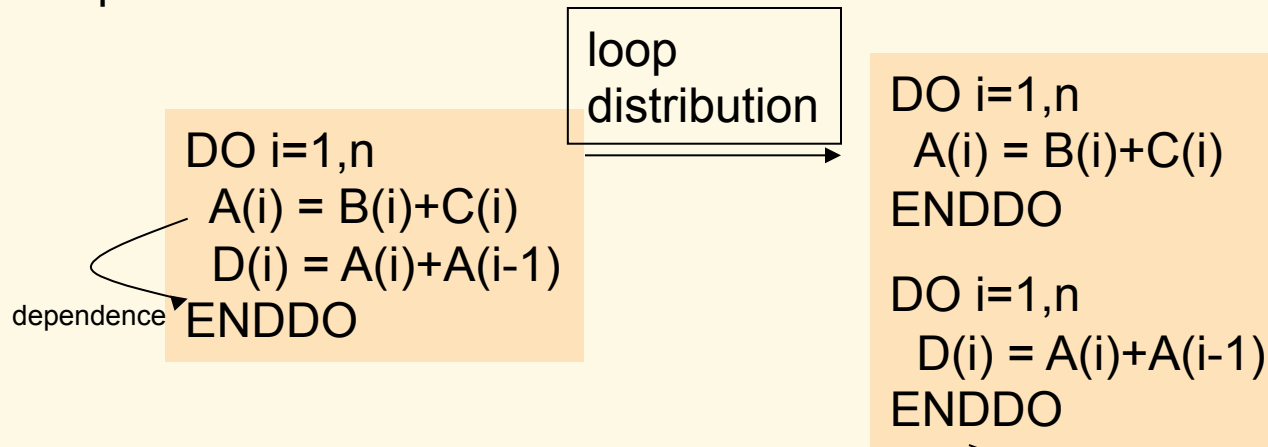
```
DO i=1,n  
  A(i) = B(i)+C(i)  →  A(1:n)=B(1:n)+C(1:n)  
ENDDO
```

```
DO i=1,n  
  A(i) = B(i)+C(i)  →  A(1:n)=B(1:n)+C(1:n)  
  C(i-1) = D(i)**2  →  C(0:n-1)=D(1:n)**2  
ENDDO
```

Here, the triplet notation means *vector operation*. Notice that this is not necessarily the same meaning as the array notation supported by some languages.

Distribution and Vectorization

The transformation done on the previous slide involves loop distribution. Loop distribution reorders computation and is thus subject to data dependence constraints.



The transformation is not legal if there is a lexical-backward dependence:

`A(1:n)=B(1:n)+C(1:n)`
`D(1:n)=A(1:n)+A(0:n-1)`

vectorization

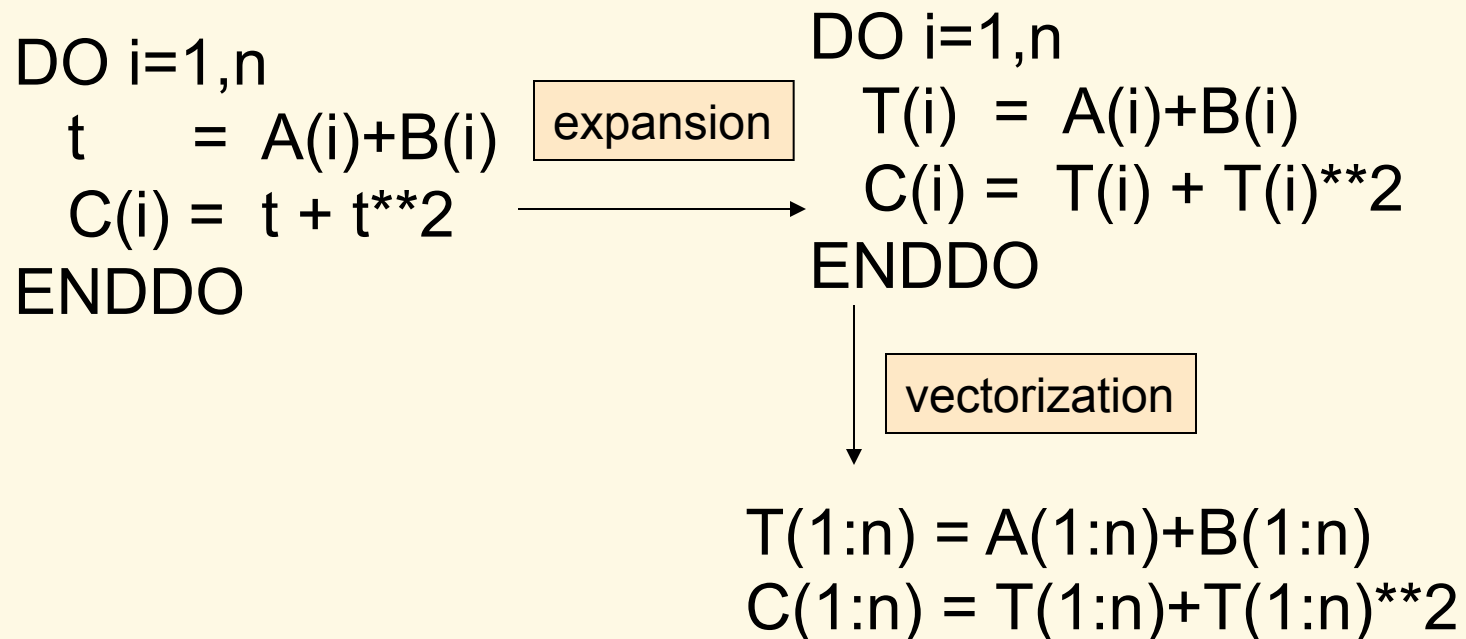
`DO i=1,n`
`A(i) = B(i)+C(i)`
`C(i+1) = D(i)**2`
`ENDDO`

loop-carried dependence

Statement reordering may help resolve the problem. However, this is not possible if there is a dependence cycle.

Vectorization Needs Expansion

... as opposed to privatization



Conditional Vectorization

```
DO i=1,n  
  IF (A(i) < 0) A(i)=-A(i)  
ENDDO
```



conditional vectorization

```
WHERE (A(1:n) < 0) A(1:n)=-A(1:n)
```

Stripmining for Vectorization

```
DO i=1,n  
  A(i) = B(i)  
ENDDO
```

stripmining

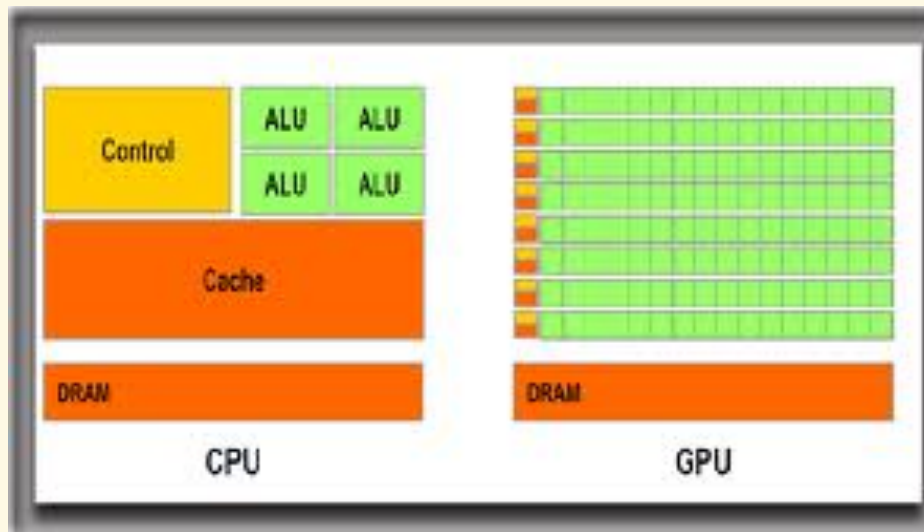
→

```
DO i1=1,n,32  
  DO i=i1,min(i1+31,n)  
    A(i) = B(i)  
  ENDDO  
ENDDO
```

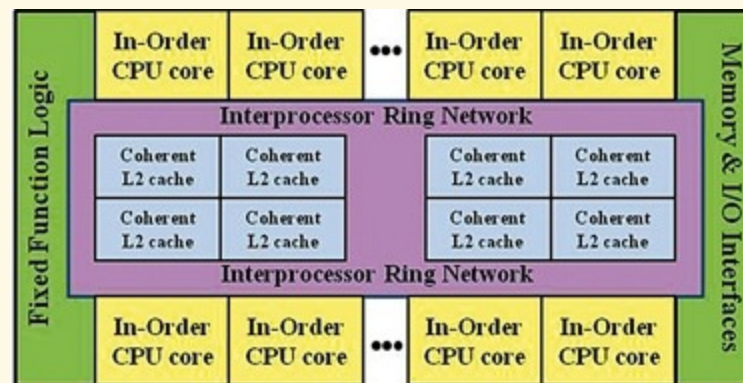
Stripmining turns a single loop into a doubly-nested loop for two-level parallelism. It also needs to be done by the code-generating compiler to split an operation into chunks of the available vector length.

Programming Accelerators

Examples of Accelerators



Nvidia
GPGPU



Intel MIC
(Xeon Phi)

Why Accelerators?

At a high level,

special-purpose architectures can execute certain program patterns faster or with less energy than general-purpose CPUs.

Examples:

- vector machines
- GRAPE (molecular-dynamics processor)
- DSP
- FPGA

Why Accelerators?

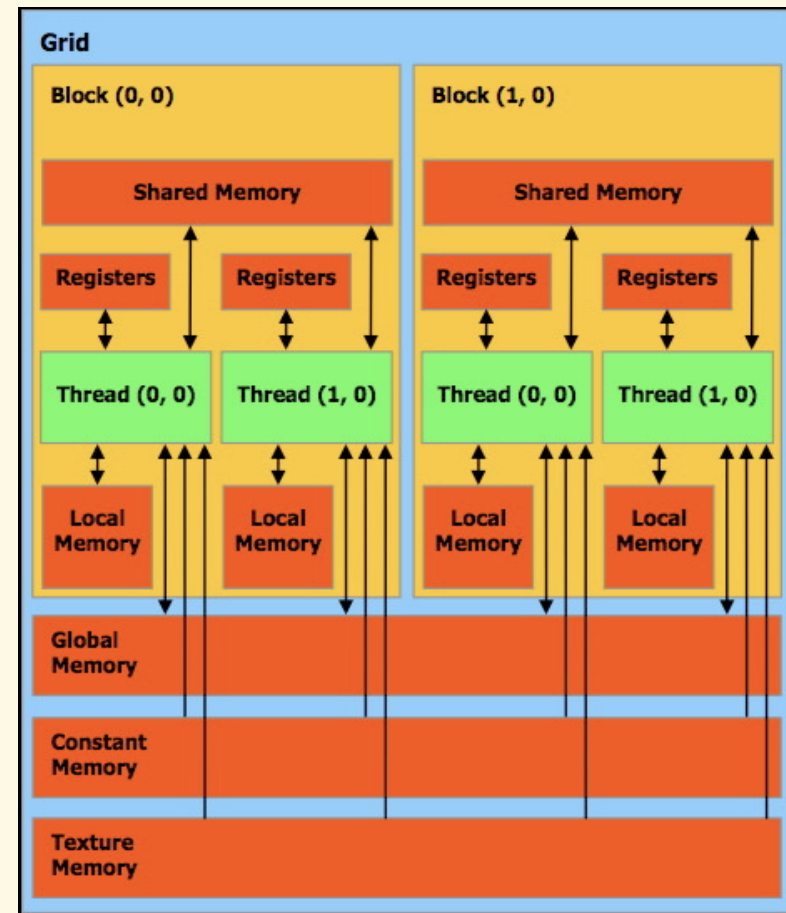
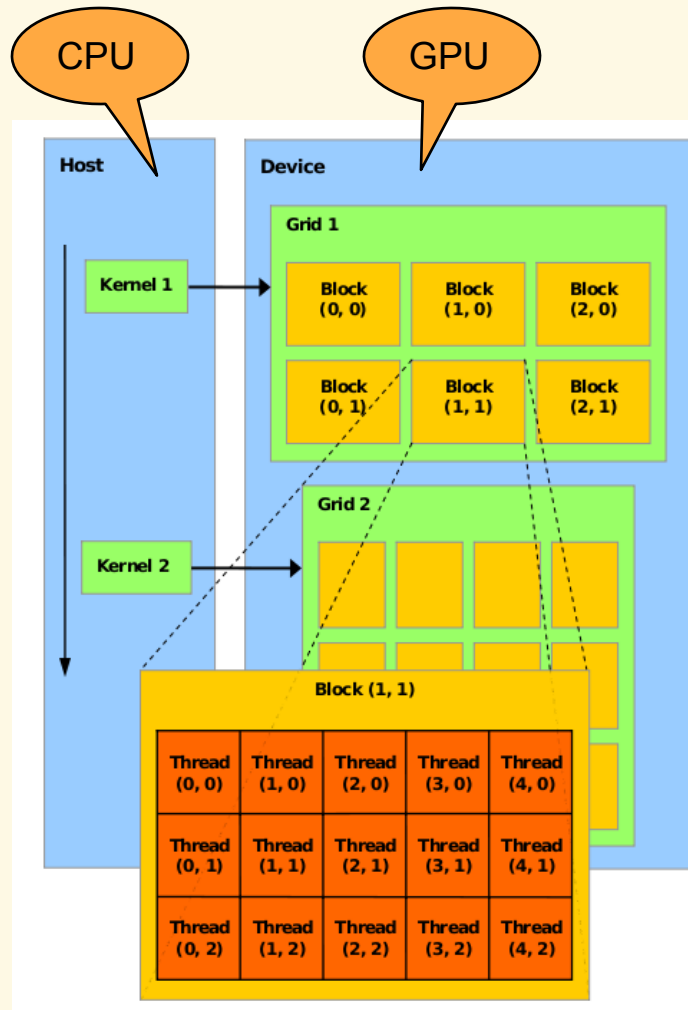
More specifically

- ◆ Heterogeneity as an opportunity
 - 1 fat processor for sequential code
 - + many lean cores for highly parallel code
- ◆ Tradeoff: fat core/large cache versus many of them
- ◆ GPUs exist anyway, use them for computation as well
- ◆ Energy argument
- ◆ high bandwidth AND low (effective) latency for certain access patterns

Accelerators Today

- ◆ GPGPUs have accelerated many applications
- ◆ Performance varies widely
- ◆ Emphasis is on science/engineering applications
- ◆ Fastest supercomputers use GPGPUs
- ◆ Nvidia GPGPUs dominate
 - key architecture features: SIMD and multithreading
- ◆ Intel's LRB did not become a product. However, MIC (Xeon Phi) has just entered the market.
- ◆ Programming productivity is poor
 - programmer needs to understand complex architecture
 - new programming constructs and techniques

CUDA Architecture



Programming Models

◆ History:

- Explicit offloading and optimization: CUDA, OpenCL, OpenACC

◆ Pure Shared Memory model:

- Compiler performs the translation (OpenMP research compiler)

◆ Shared Memory plus directives:

- OpenACC, OpenMP 4.0, OpenMP (research compiler)

Important Accelerator Programming Techniques

- ◆ defining kernels for offloading
- ◆ creating coalesced accesses (loop interchange, coalescing, data transpose)
 - this is the most-important optimization (note, this was true for vector machines as well)
- ◆ managing copy-in/out
- ◆ data placement in accelerator memory hierarchy
- ◆ defining thread block size

Programming Complexity Comparison (Jacobi)

OpenMP code

```
float a[SIZE_2][SIZE_2];
float b[SIZE_2][SIZE_2];
int main (int argc, char *argv[]) {
    int i, j, k;
    for (k = 0; k < ITER; k++) {
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                a[i][j] = (b[i - 1][j] + b[i + 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0f;
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                b[i][j] = a[i][j];
    }
    return 0;
}
```

CUDA code

```
float a[SIZE_2][SIZE_2]; float b[SIZE_2][SIZE_2];
__global__ void main_kernel0(float a[SIZE_2][SIZE_2], float b[SIZE_2][SIZE_2]) {
    int i,j; int _bid = (blockIdx.x+(blockIdx.y*gridDim.x)); int _gtid = (threadIdx.x+_bid*blockDim.x);
    i=_gtid+1;
    if (i<=SIZE {
        for (j=1; j<=SIZE j++)
            a[i][j]=(b[(i-1)][j]+b[(i+1)][j]+b[i][(j-1)]+b[i][(j+1)])/4.0f;
    }
}
__global__ void main_kernel1(float a[SIZE_2][SIZE_2], float b[SIZE_2][SIZE_2]) {
    int i,j; int _bid = (blockIdx.x+(blockIdx.y*gridDim.x)); int _gtid = (threadIdx.x+_bid*blockDim.x);
    i=_gtid+1;
    if (i<=SIZE {
        for (j=1; j<=SIZE j++)
            b[i][j]=a[i][j];
    }
}
int main(int argc, char * argv[]) {
    int i,j,k; float * gpu_a; float * gpu_b;
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);
    CUDA_SAFE_CALL(cudaMalloc((void * *) (& gpu_a), gpuBytes));
    dim3 dimBlock0(BLOCK_SIZE, 1, 1); dim3 dimGrid0(NUM_BLOCKS, 1, 1);
    CUDA_SAFE_CALL(cudaMemcpy(gpu_a, a, gpuBytes, cudaMemcpyHostToDevice));
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);
    CUDA_SAFE_CALL(cudaMalloc((void * *) (& gpu_b), gpuBytes));
    CUDA_SAFE_CALL(cudaMemcpy(gpu_b, b, gpuBytes, cudaMemcpyHostToDevice));
    dim3 dimBlock1(BLOCK_SIZE, 1, 1); dim3 dimGrid1(NUM_BLOCKS, 1, 1);
    for (k=0; k<ITER; k++) {
        main_kernel0<<<dimGrid0, dimBlock0, 0, 0>>>((float *) [SIZE_2])gpu_a, (float *) [SIZE_2])gpu_b);
        main_kernel1<<<dimGrid1, dimBlock1, 0, 0>>>((float *) [SIZE_2])gpu_a, (float *) [SIZE_2])gpu_b);
    }
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);
    CUDA_SAFE_CALL(cudaMemcpy(b, gpu_b, gpuBytes, cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL(cudaFree(gpu_b));
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);
    CUDA_SAFE_CALL(cudaMemcpy(a, gpu_a, gpuBytes, cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL(cudaFree(gpu_a));
    fflush(stdout); fflush(stderr); return 0;
}
```

OpenMP Version of Jacobi

```
float a[SIZE_2][SIZE_2];
float b[SIZE_2][SIZE_2];
int main (int argc, char *argv[]) {
    int i, j, k;
    for (k = 0; k < ITER; k++) {
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                a[i][j] = (b[i - 1][j] + b[i + 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0f;
        #pragma omp parallel for private(i, j)
        for (i = 1; i <= SIZE; i++)
            for (j = 1; j <= SIZE; j++)
                b[i][j] = a[i][j];
    }
    return 0;
}
```

CUDA Version of Jacobi

```
int main(int argc, char * argv[]) { GPU Memory Allocation and Data Transfer to GPU
```

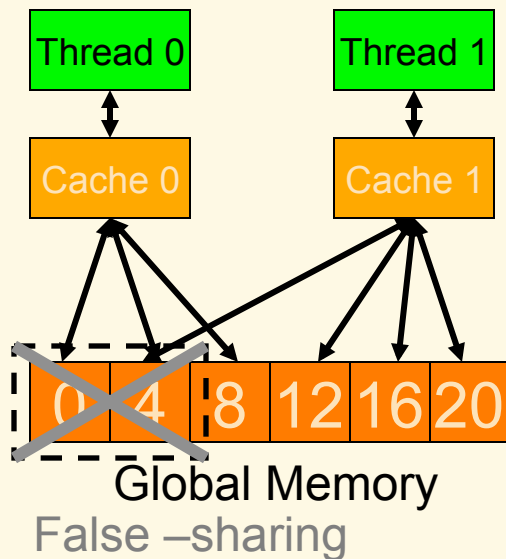
```
    int i,j,k; float * gpu__a; float * gpu__b;  
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);  
    CUDA_SAFE_CALL(cudaMalloc(((void * *)(& gpu__a)), gpuBytes));  
    CUDA_SAFE_CALL(cudaMemcpy(gpu__a, a, gpuBytes, cudaMemcpyHostToDevice));  
    gpuBytes=((SIZE_2*SIZE_2)*sizeof (float));  
    CUDA_SAFE_CALL(cudaMalloc(((void * *)(& gpu__b)), gpuBytes));  
    CUDA_SAFE_CALL(cudaMemcpy(gpu__b, b, gpuBytes, cudaMemcpyHostToDevice));  
    dim3 dimBlock0(BLOCK_SIZE, 1, 1); dim3 dimGrid0(NUM_BLOCKS, 1, 1);  
    dim3 dimBlock1(BLOCK_SIZE, 1, 1); dim3 dimGrid1(NUM_BLOCKS, 1, 1);  
    for (k=0; k<ITER; k ++ ) {  
        main_kernel0<<<dimGrid0, dimBlock0, 0, 0>>>((float (*)[SIZE_2])gpu__a,  
            (float (*)[SIZE_2])gpu__b);  
        main_kernel1<<<dimGrid1, dimBlock1, 0, 0>>>((float (*)[SIZE_2])gpu__a,  
            (float (*)[SIZE_2])gpu__b);  
    } GPU Kernel Execution
```

```
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);  
    CUDA_SAFE_CALL(cudaMemcpy(b, gpu__b, gpuBytes, cudaMemcpyDeviceToHost));  
    gpuBytes=(SIZE_2*SIZE_2)*sizeof (float);  
    CUDA_SAFE_CALL(cudaMemcpy(a, gpu__a, gpuBytes, cudaMemcpyDeviceToHost));  
    CUDA_SAFE_CALL(cudaFree(gpu__b));  
    CUDA_SAFE_CALL(cudaFree(gpu__a));  
    fflush(stdout); fflush(stderr); return 0;
```

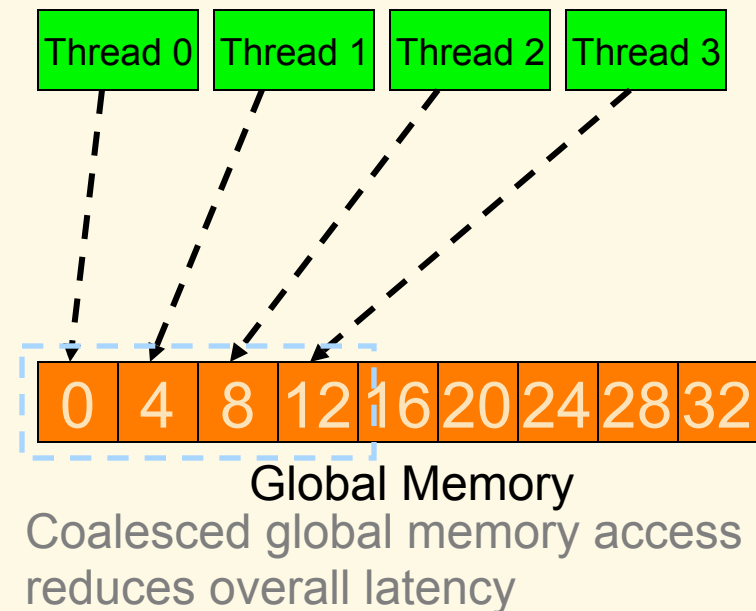
```
}
```

Data Transfer Back To CPU and GPU Memory Deallocation

Intra-Thread vs. Inter-Thread Locality



Common CPU Memory Model



CUDA Memory Model

- ◆ Intra-thread locality is beneficial to both OpenMP and CUDA model.
- ◆ Inter-thread locality plays a critical role in CUDA model.

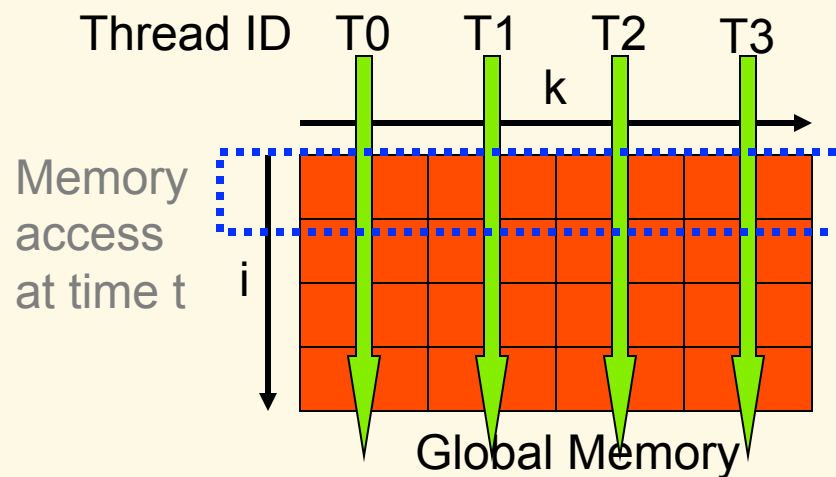
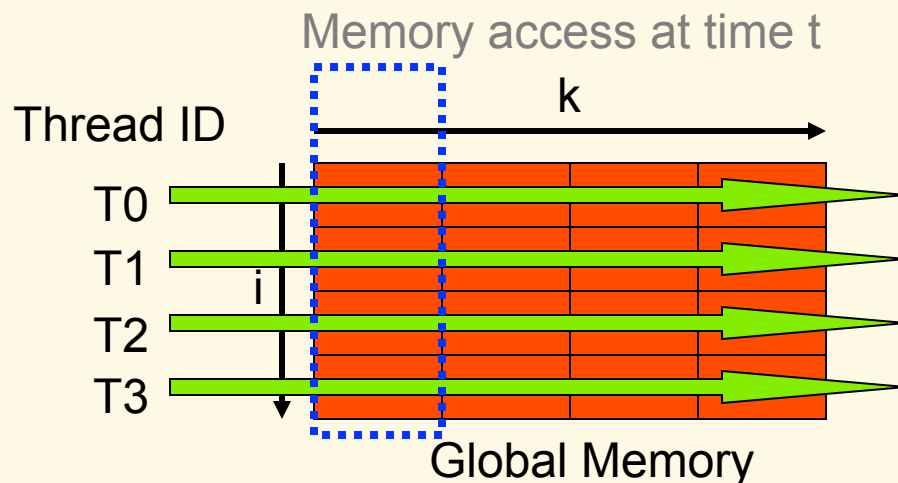
Effective GPGPU Programming Techniques: Parallel Loop-Swap

```
#pragma omp parallel for  
for(i=0; i< N; i++)  
    for(k=0; k<N; k++)  
        A[i][k] = B[i][k];
```

Input OpenMP code

```
#pragma omp parallel for  
    schedule(static, 1)  
for(k=0; k<N; k++)  
    for(i=0; i<N; i++)  
        A[i][k] = B[i][k];
```

Optimized OpenMP code



Effective GPGPU Programming Techniques:

Loop Collapsing

```
#pragma omp parallel for
for(i=0; i<n_rows; i++)
    for(k=rptr[i]; k<rptr[i+1]; k++)
        w[i] += A[k]*p[col[k]];
```

Input OpenMP code

```
#pragma omp parallel
#pragma omp for collapse(2)
    schedule(static, 1)
for(i=0; i<n_rows; i++)
    for(k=rptr[i]; k<rptr[i+1]; k++)
        w[i] += A[k]*p[col[k]];
```

Optimized OpenMP code

```

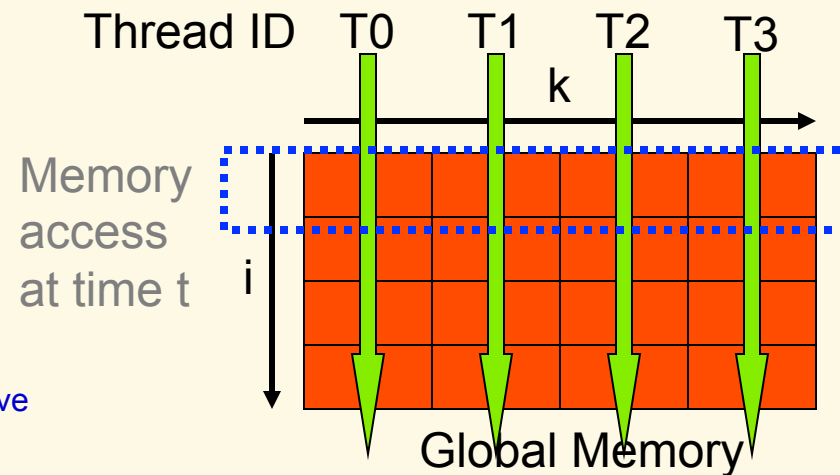
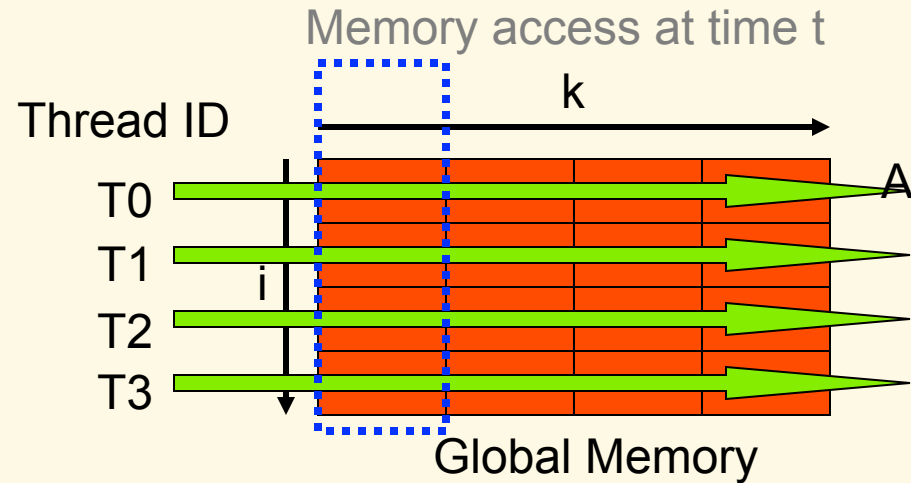
// Collapsed loop
If( tid1 < rptr[n_rows])
    l_w[tid1] = A[tid1]*p[col[tid1]];

// Reduction loop
If( tid2 < n_rows )
    for( k=rptr[tid2]; k<rptr[tid2+1]; k++ )
        w[tid2] += l_w[k];
(c) GPU code
```

Effective GPGPU Programming Techniques: Matrix-Transpose

```
#pragma omp parallel for
  schedule(static, 1)
  transpose(A)1
for(i=0; i< N; i++) {
  for(k=0; k<M; k++)
    ... = A[i,k];
}
```

¹OpenMP standard does not include a transpose directive



Effective GPGPU Programming Techniques: Managing Copy-in/out

1. Eliminate copy-out of data that are not live out
2. Leave data that is needed in future kernel invocations in the device memory
 - eliminate copy-out if not needed on the CPU side
3. Narrow the copy-in data range to the minimum needed
4. Copy-in early (overlap copy-in with execution of previous kernel)
5. Pipeline copy-in and execution within same kernel.

Note that techniques 2,4,5 increase the demand on device memory

Effective GPGPU Programming Techniques: Setting Thread Block Size

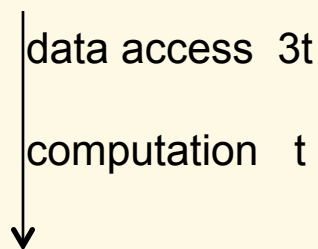
```
for i=1,n  
  ... = a[i]
```

there can be max n (logical) threads, however:

Choosing a threadblock size $< \text{max}$ may be preferable.

- ◆ Shared resources: fewer threads incur fewer resource conflicts
- ◆ Multithreading: If a thread (block) stalls on a memory transfer, a different thread (block) becomes active.

thread

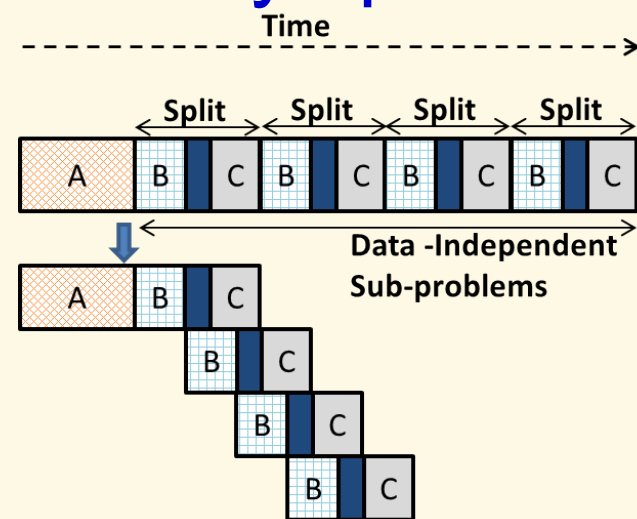


≥ 3 thread blocks are needed, in this example, to overcome the memory latency

Note that different vendors use the terms thread and block differently

Effective GPGPU Programming Techniques: Memory Management

- ◆ Accelerators may not provide virtual memory management
- ◆ Blocking/stripmining may be needed to fit data within available memory space
- ◆ Side benefit of blocking:
overlapping copy-in/out
and computation



SMP Programming Errors

- ◆ Shared memory parallel programming is a mixed bag:
 - It saves the programmer from having to map data onto multiple processors. In this sense, its much easier.
 - It opens up a range of new errors coming from unanticipated shared resource conflicts.

2 major SMP errors

◆ Race Conditions

- ▼ The outcome of a program depends on the detailed timing of the threads in the team.

◆ Deadlock

- ▼ Threads lock up waiting on a locked resource that will never become free.

◆ Livelock

- ▼ A termination condition is never reached

Race Conditions

C\$OMP PARALLEL SECTIONS

$$A = B + C$$

C\$OMP SECTION

$$B = A + C$$

C\$OMP SECTION

$$C = B + A$$

C\$OMP END PARALLEL SECTIONS

- ◆ The result varies unpredictably based on detailed order of execution for each section.
- ◆ Wrong answers produced without warning!

Race Conditions:

A complicated (silly?) solution

```
ICOUNT = 0
C$OMP PARALLEL SECTIONS
  A = B + C
  ICOUNT = 1
C$OMP FLUSH ICOUNT
C$OMP SECTION
1000 CONTINUE
C$OMP FLUSH ICOUNT
  IF(ICOUNT .LT. 1) GO TO 1000
  B = A + C
  ICOUNT = 2
C$OMP FLUSH ICOUNT
C$OMP SECTION
2000 CONTINUE
C$OMP FLUSH ICOUNT
  IF(ICOUNT .LT. 2) GO TO 2000
  C = B + A
C$OMP END PARALLEL SECTIONS
```

- ◆ In this example, we choose the assignments to occur in the order A, B, C.
 - ICOUNT forces this order.
 - FLUSH so each thread sees updates to ICOUNT - NOTE: you need the flush on each read and each write.

A More Subtile Race Condition

```
C$OMP PARALLEL SHARED (X)
C$OMP& PRIVATE(ID)

C$OMP PARALLEL DO REDUCTION(+:X)
  DO 100 I=1,100
    TMP = WORK(I)
    X = X + TMP
100 CONTINUE
C$OMP END DO

C$OMP END PARALLEL
```

- ◆ The result varies unpredictably because access to shared variable `TMP` is not protected.
- ◆ Wrong answers produced without warning!
- ◆ The user probably wanted to make `TMP` private.

Avoiding Race Conditions

- ◆ Easiest solution: don't access any variable that is written by another parallel thread.
 - this is always the case for fully parallel loops
- ◆ More complicated: if data dependences are unavoidable, synchronize them properly.
 - Be aware that you reduce the performance
- ◆ Avoid: creating your own synchronization by “waiting for a flag set by the other thread”.
 - use provided synchronization primitives instead
- ◆ (Desirable) race conditions seen in real programs:
 - parallel shuffle
 - parallel search

Deadlock

```
CALL OMP_INIT_LOCK (LCKA)
C$OMP PARALLEL SECTIONS
C$OMP SECTION
CALL OMP_SET_LOCK(LCKA)
IVAL = DOWORK()
IF (IVAL .GT. TOL) THEN
CALL ERROR (IVAL)
ELSE
CALL OMP_UNSET_LOCK (LCKA)
ENDIF
ENDIF
C$OMP SECTION
CALL OMP_SET_LOCK(LCKA)
CALL USE_B_and_A (RES)
CALL OMP_UNSET_LOCK(LCKA)
C$OMP END SECTIONS
```

- ◆ If A is locked in the first section and the if statement branches around the unset lock, threads running the other sections deadlock waiting for the lock to be released.
- ◆ Make sure you release your locks. Always, even in error situations!

Livelock

```
C$OMP PARALLEL PRIVATE(ID)
  ID = OMP_GET_THREAD_NUM()
  N = OMP_GET_NUM_THREADS()
1000 CONTINUE
  PHASES[ID] = UPDATE(U, ID)
C$OMP SINGLE
  RES = MATCH (PHASES, N)
C$OMP END SINGLE
  IF (RES*RES .LT. TOL) GO TO 2000
  GO TO 1000
2000 CONTINUE
C$OMP END PARALLEL
```

- ◆ This shows a race condition and a livelock.
- ◆ If the square of RES is never smaller than TOL, the program spins endlessly in Livelock.

Livelock

```
    ICOUNT = 0
C$OMP PARALLEL PRIVATE (ID)
    ID = OMP_GET_THREAD_NUM()
    N = OMP_GET_NUM_THREADS()
1000 CONTINUE
    PHASES[ID] = UPDATE(U, ID)
C$OMP BARRIER
C$OMP SINGLE
    RES = MATCH (PHASES, N)
    ICOUNT = ICOUNT + 1
C$OMP END SINGLE
    IF (RES*RES .LT. TOL) GO TO 2000
    IF (ICOUNT .GT. MAX) GO TO 2000
    GO TO 1000
2000 CONTINUE
C$OMP END PARALLEL
```

◆ Solution:

- Fix the race with a barrier before the single. This may fix the MATCH operation and fix the livelock error.
- Decide on a maximum number of iterations, and use a loop with that number rather than an infinite loop.

SMP (OpenMP) Error Advice

- ◆ Are you using threadsafe libraries?
- ◆ I/O inside a parallel region can interleave unpredictably.
- ◆ Make sure you understand what your constructors are doing with private objects.
- ◆ Watch for private variables masking globals.
- ◆ Understand when shared memory is coherent. When in doubt, use FLUSH.
- ◆ NOWAIT removes implied barriers.

Navigating through the Danger Zones

- ◆ Option 1: Analyze your code to make sure every semantically permitted interleaving of the threads yields the correct results.
 - This can be prohibitively difficult due to the explosion of possible interleavings.
 - Tools like Intel's Thread Checker (Assure) can help.

Navigating through the Danger Zones

- ◆ Option 2: Write SMP code that is portable and equivalent to the sequential form.
 - Use a safe subset of OpenMP.
 - Follow a set of “rules” for Sequential Equivalence.

Portable Sequential Equivalence

- ◆ What is Portable Sequential Equivalence (PSE)?
 - ▼ A program is sequentially equivalent if its results are the same with one thread and many threads.
 - ▼ For a program to be portable (i.e. runs the same on different platforms/compilers) it must execute identically when the OpenMP constructs are used or ignored.

Portable Sequential Equivalence

◆ Advantages of PSE

- ▼ A PSE program can run on a wide range of hardware and with different compilers - minimizes software development costs.
- ▼ A PSE program can be tested and debugged in serial mode with off-the-shelf tools - even if they don't support OpenMP.

2 Forms of Sequential Equivalence

- ◆ Two forms of Sequential equivalence based on what you mean by the phrase “equivalent to the single threaded execution”:
 - ▼ Strong SE: bitwise identical results.
 - ▼ Weak SE: equivalent mathematically but due to quirks of floating point arithmetic, not bitwise identical.

Strong Sequential Equivalence: Rules

- Control data scope with the base language
 - ▼ Avoid the data scope clauses, except...
 - ▼ Only use private for scratch variables local to a block (e.g. temporaries or loop control variables) whose global initialization don't matter.
- Locate all cases where a shared variable written by one thread is accessed (read or written) by another threads.
 - ▼ All accesses to the variable must be protected.
 - ▼ If multiple threads combine results into a single value, enforce sequential order.
 - ▼ Do not use the reduction clause.

Strong Sequential Equivalence: example

```
C$OMP PARALLEL PRIVATE(I, TMP)
```

```
C$OMP DO ORDERED
```

```
  DO 100 I=1,NDIM
```

```
    TMP =ALG_KERNEL(I)
```

```
C$OMP ORDERED
```

```
  CALL COMBINE (TMP, RES)
```

```
C$OMP END ORDERED
```

```
100 CONTINUE
```

```
C$OMP END PARALLEL
```

- ◆ Everything is shared except I and TMP. These can be private since they are not initialized and they are unused outside the loop.
- ◆ The summation into RES occurs in the sequential order so the result from the program is bitwise compatible with the sequential program.
- ◆ Problem: Can be inefficient if threads finish in an order that's greatly different from the sequential order.

Weak Sequential equivalence

- ◆ For weak sequential equivalence only mathematically valid constraints are enforced.
 - ▼ Computer floating point arithmetic is not associative and not commutative.
 - ▼ In most cases, no particular grouping of floating point operations is mathematically preferred so why take a performance hit by forcing the sequential order?
 - In most cases, if you need a particular grouping of floating point operations, you have a bad algorithm.
- ◆ How do you write a program that is portable and satisfies weak sequential equivalence?
 - Follow the same rules as the strong case, but relax sequential ordering constraints.

Weak equivalence: example

```
C$OMP PARALLEL PRIVATE(I, TMP)
C$OMP DO
    DO 100 I=1,NDIM
        TMP =ALG_KERNEL(I)
C$OMP CRITICAL
    CALL COMBINE (TMP, RES)
C$OMP END CRITICAL
100 CONTINUE

C$OMP END PARALLEL
```

- ◆ The summation into RES occurs one thread at a time, but in any order so the result is not bitwise compatible with the sequential program.
- ◆ Much more efficient, but some users get upset when low order bits vary between program runs.

Sequential Equivalence isn't a Silver Bullet

```
C$OMP PARALLEL
C$OMP& PRIVATE(I, ID, TMP, RVAL)
  ID = OMP_GET_THREAD_NUM()
  N = OMP_GET_NUM_THREADS()
  RVAL = RAND ( ID )
C$OMP DO
  DO 100 I=1,NDIM
    RVAL = RAND (RVAL)
    TMP =RAND_ALG_KERNEL(RVAL)
C$OMP CRITICAL
  CALL COMBINE (TMP, RES)
C$OMP END CRITICAL
100 CONTINUE
C$OMP END PARALLEL
```

- ◆ This program follows the weak PSE rules, but its still wrong.
- ◆ In this example, RAND() may not be thread safe. Even if it is, the pseudo-random sequences might overlap thereby throwing off the basic statistics.

Map Reduce

◆ Wikipedia:

MapReduce is

- a programming model for processing large data sets, and
- the name of an implementation of the model by Google.

MapReduce is typically used to do distributed computing on clusters of computers.

MapReduce libraries have been written in many programming languages. A popular, free implementation is Apache Hadoop.

Why MapReduce?

“MapReduce provides regular programmers the ability to produce parallel distributed programs much more easily, by requiring them to write only the simpler Map() and Reduce() functions, which focus on the logic of the specific problem at hand”

However,

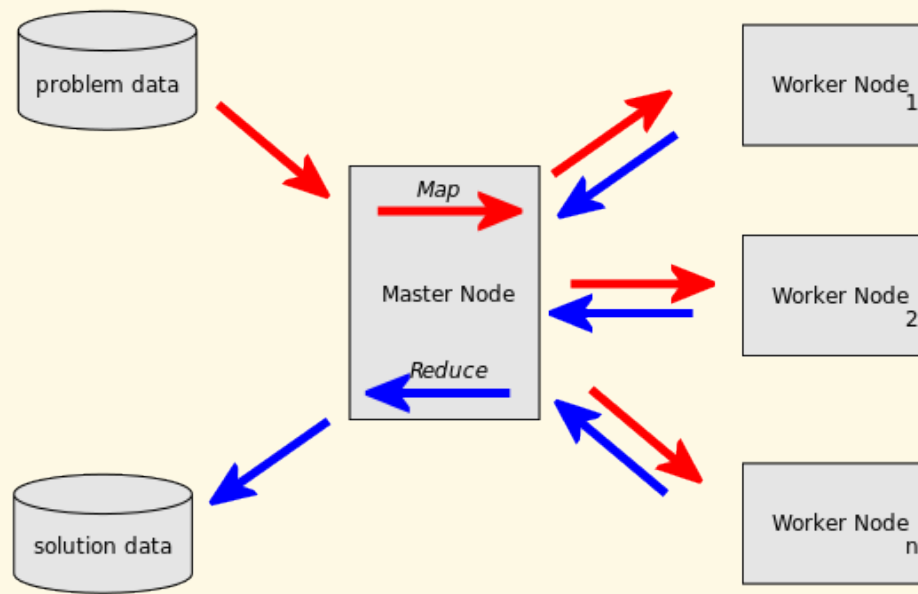
you need to have a good understanding how map, reduce, and the overall system interact.

Many problems can be expressed as such a two-step algorithm

Map step

“The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.”

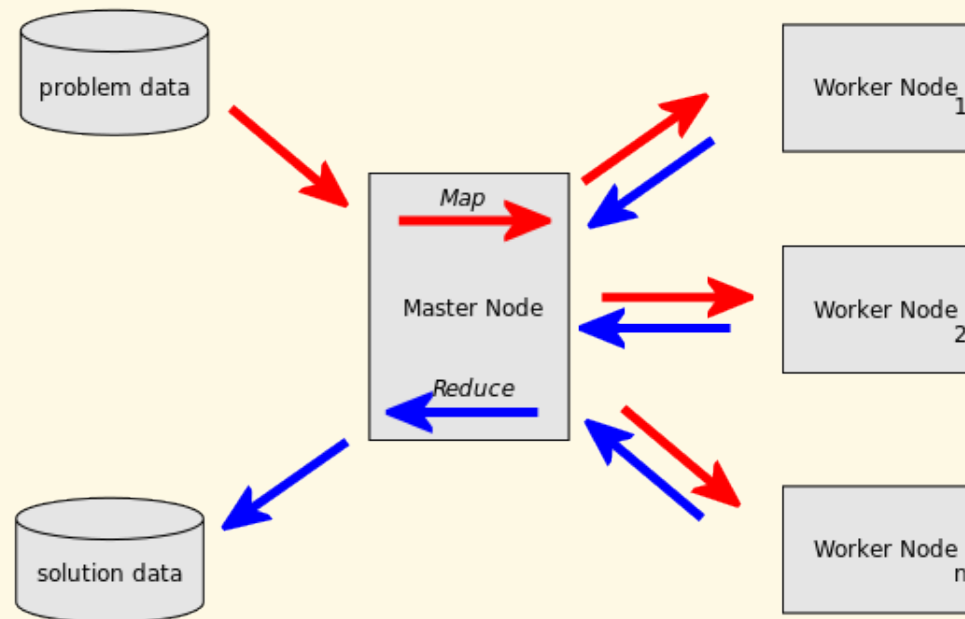
- Map is performed fully parallel on each subproblem



Reduce step

“ The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve.”

- Reduce is not fully parallel, but may be expressed as a sequence of parallel combine steps.



A Simple Example: Parallel Reduction Expressed as MapReduce

- ◆ Problem: sum all n elements an an array
- ◆ Master: splits array into #processor parts; calls map for each processor
- ◆ Map: sums all elements or the assigned array part; returns partial sum
- ◆ Reduce: receives partial sums; returns their sum

Processors can assume the role of Master (e.g. if the assigned part is larger than a threshold) and engage additional processors, in turn.

=> leads to a combining tree for the summation.

A 5-step Process

1. Prepare the Map() input – the "MapReduce system" designates Map processors, assigns the K1 input key value each processor would work on, and provides that processor with all the input data associated with that key value.
2. Run the user-provided Map() code – Map() is run exactly once for each K1 key value, generating output organized by key values K2. *MAP*
3. "Shuffle" the Map output to the Reduce processors – the MapReduce system designates Reduce processors, assigns the K2 key value each processor would work on, and provides that processor with all the Map-generated data associated with that key value.
4. Run the user-provided Reduce() code – Reduce() is run exactly once for each K2 key value produced by the Map step. *REDUCE*
5. Produce the final output – the MapReduce system collects all the Reduce output, and sorts it by K2 to produce the final outcome.

A More Advanced Example: Word Count

```
function Map(String name, String document):  
    count each word in the document  
    return the map < word, #occurrences >
```

```
function Reduce(String word, List #occurrences):  
    sum =  $\sum$  #occurrences  
    return < word, sum >
```

- Map returns multiple results, each annotated with a key. In our example, the key identifies a specific word.
- The shuffle function combines the values (#occurrences) for each key returned by a map function into a list and calls Reduce(key,list)
 - shuffle can take a substantial amount of time. It is implemented by the system; the user only needs to write Map and Reduce.

Other MapReduce Problems/ Applications

- ◆ searching
- ◆ sorting
- ◆ web link-graph reversal
- ◆ web access log stats
- ◆ inverted index construction
- ◆ document clustering
- ◆ machine learning
- ◆ statistical machine translation

MapReduce has also been demonstrated on some linear algebra problems, such as matrix multiply

“Basically wherever you see linear algebra (matrix/vector operations) you can apply Map Reduce”

OTHER PROGRAMMING MODELS, LANGUAGES, CONCEPTS

Data Flow

Parallelism is derived from the availability of data in a data flow graph.

◆ Joule (1996):

- Concurrent dataflow programming language for building distributed applications. The order of statements within a block is irrelevant to the operation of the block. Statements are executed whenever possible, based on their inputs. Everything in Joule happens by sending messages. There is no control flow. Instead, the programmer describes the flow of data, making it a data flow programming language.

◆ SISAL (1993):

- Uses and implements single-assignment concepts. Produces a data flow graph. (Single assignment removes anti and output dependences, which exposes maximum parallelism)

◆ StarS (2009):

- Hierarchical task-based programming with StarSs, Badia, Ayguade, Labarta (2009)

Global Address Space Languages (GAS)

- ◆ UPC, co-array Fortran, HPF
- ◆ Languages for distributed-memory machines
- ◆ The user sees a shared address space
- ◆ There are constructs for data distribution. They can be explicit, user-provided data-distribution directives or array dimensions that indicate the processor ID.
- ◆ Compilers translate the GAS program into a message-passing form.

Non-Blocking Operations

- ◆ Fundamental mechanism for overlapping communication and (multiple) operations.

```
a := b  
...  
synch()
```

the computation of the value b and transfer to a is initiated, and completed at the synch() point. It overlaps with the execution of the “...” statements.

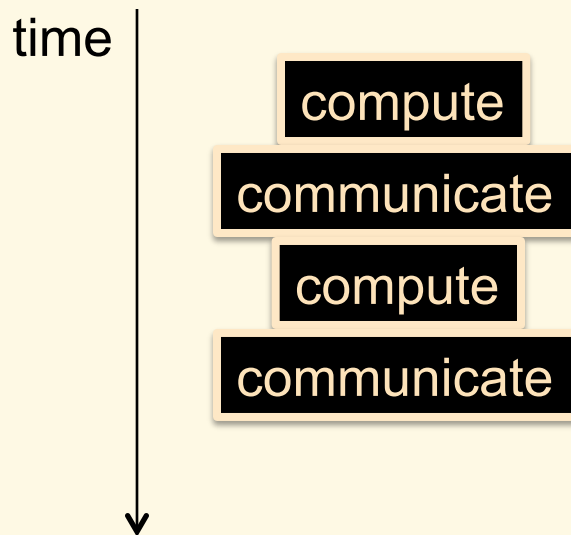
- ◆ If b is a simple value, this is essentially the same as prefetch
- ◆ If b is an operation, this involves the spawning of a parallel task

CSP – Communicating Sequential Processes (Hoare, 1978)

- ◆ one of the earliest concepts for expressing parallel activities
- ◆ center is the process
- ◆ synchronization and communication constructs are very important
 - semaphores, monitors, locks, etc.
- ◆ OCCAM/Transputer

BSP – Block Synchronous Parallelism

- ◆ Structuring a parallel computation into phases of full parallelism followed by communication phases.



Parallel models differ in the primary concerns they require the user to deal with

- ◆ data flow: focus on needed and produced data by activities
- ◆ control flow: how control flows and transfers
 - focus on starting and ending parallel activities
- ◆ control transfer: focus on how cpu resources are assigned to activities
- ◆ messaging: focus on communication (and implied synchronization) between parallel activities

High-level Problem Classes

- ◆ Fixed problem to be solved in shortest possible time.
 - HPC applications
- ◆ On-demand services to be made available on a continual basis.
 - Operating systems, internet services
- ◆ Continuous data streams to be processed.
 - Image processing

Wikipedia:

Concurrent and parallel programming languages

- ◆ Actor Model
- ◆ Coordination Languages
- ◆ CSP-based
- ◆ Dataflow
- ◆ Distributed Event-driven and hardware description
- ◆ Functional
- ◆ GPU languages
- ◆ Logic programming
- ◆ Multi-threaded
- ◆ Object oriented
- ◆ PGAS
- ◆ Unsorted

Wikipedia:

Parallel programming models

- ◆ Process interaction
- ◆ Shared memory
- ◆ Message passing
- ◆ Implicit parallelism
- ◆ Problem decomposition
- ◆ Task parallelism
- ◆ Data parallelism