Address Buffer linearly

count = 0;
iend = ihi + 1;
jend = jhi + 1;
for (i=ilo; i<iend; i++) {
    for (j=jlo; j<jend; j++) {
        buffer[count] = 
            a*(double)i + b*(double)j + c;
        count++;
    }
}

Address Buffer linearly

count = 0;
for (i=ilo; i<=ihi; i++) {
    for (j=jlo; j<=jhi; j++) {
        buffer[count] =
            a*(double)i + b*(double)j + c;
        count++;
    }
}


Replicated Data Algorithms

• Our first parallel programming model.
• Complete data structures are replicated on all nodes.
• Some data structures may contain all the data
• Others may only contain partial contributions to the entire structure.
• Work is partitioned based on logical processor or thread Identification.
• The replication serves as the Shared-Memory similarity.
Consider matrix generation

allocate storage for A on All nodes;
if (myid == master)
  for(i=0;i<ranki;i++)
    for(j=0;j<rankj;j++)
      A[i][j] = genA( );

Broadcast data from 0 to others

0 Has data
1 Has storage
2 Has storage
3 Has storage

0 Has data
1 Has data
2 Has data
3 Has data

1/26/2005  ComS 425  Spring 2005  5 of 44: Lecture 7
Consider Matrix Multiply

\[
\begin{align*}
\text{pass} &= -1; \\
\text{for} (i=0; i<\text{ranki}; i++) \{ \\
\hspace{1em} \text{pass} &\text{ ++;} \\
\hspace{1em} \text{if (} (\text{pass} \% \text{Nproc}) == \text{myID}) \{ \\
\hspace{2em} \text{for} (j=0; j<\text{rankj}; j++) \\
\hspace{3em} \text{for} (k=0; k<\text{rankk}; k++) \{ \\
\hspace{4em} c[i][j] &\text{ += a[i][k]*b[k][j];} \\
\hspace{3em} \} \\
\hspace{2em} \} \\
\} \\
\end{align*}
\]
Determining the Granularity

• All nodes execute every iteration of the pre-pass-test loops
  ➢ Replication or redundant parallelism

• If the pass-test is coded outside the i loop then only one node is used.
  ➢ At most 1 processor; boring 😊

• If the pass-test is coded for the j loop then there are ranki tasks of size (rankj*rankk)
  ➢ Previous slide!!
  ➢ At most ranki processors
Consider Matrix Multiply Again

```c
pass = -1;
for(i=0;i<ranki;i++) {
    for(j=0;j<rankj;j++) {
        pass ++;
        if ((pass%Nproc) == myID) {
            for(k=0;k<rankk;k++) {
                c[i][j] += a[i][k]*b[k][j];
            }
        }
    }
}
```
Determining the Granularity Again

• All nodes execute every iteration of the pre-pass-test loops
  ➢ Replication or redundant parallelism
• If the pass-test is coded outside the i loop then only one node is used.
  ➢ At most 1 processor; boring 😊
• If the pass-test is coded for the j loop then there are ranki tasks of size (rankj*rankk)
  ➢ Previous slide!!
  ➢ At most ranki processors

• If the pass-test is coded for the k loop then there are ranki*rankj tasks of size rankk (previous slide)
  ➢ At most ranki*rankj processors
Question

• What is the best for all situations?
• This is why there is sometimes a need to code multiple implementations of the same algorithm.
• The granularity is related to the number of nodes.
• The application code has to match the granularity to the run-time situation.
  ➢ Fine granularity on a poor communication sub-system will give you poor performance!
Multithread Programming

- Multithreading (MT) is a technique that allows one program to do multiple tasks concurrently.
  - A technique that has been around since the 1970s.
  - Languages such as Ada implement tasks as multiple threads.
  - The MT programming paradigm became accepted and standardized in the 1990s.
    - In 1991 only a few OSs had a user level threads library. (Solaris)
    - In 1997 most had a threads library available to users.
  - The SMP technology was really a boon to the threads programming paradigm.
    - Combined concurrency with parallelism.
  - There is a lot of interaction between MT and OSs.
MT Examples

• What kind of programs are MT?
  ➢ Database servers
  ➢ Finance modeling and prediction.
  ➢ Chess program
  ➢ X-Windows applications
  ➢ Operating Systems
    ▪ Windoze
    ▪ Unix (Linux and FreeBSD too).
  ➢ HTTP servers!!
Programs

• Have
  - Code
    - a series of instructions
  - Data
    - Global and local types
  - Program Counter
    - pointer to current instruction
  - Stack Pointer
    - Local data and return addresses

• Interact with the operating system via a process entity.
Processes

- OSs (e.g., the kernel) have control over I/O, memory, Processors, network interfaces etc.

- For a multitasking operating system, multiple processes need access to the same resources.

- The kernel has to keep track of the process structure.

- The OS must manage switching between available processes to run.
  - Unix Time Slice algorithm
    - Roll in and Roll out.
View of a Process

User Space

Process

Stack

Global Data

Local Data

Kernel Space

Code

Data

PC

Code

Data
View of Multiple Processes

User Space

Kernel Space

Kernel Space
View of Threads

User Space

Kernel Space

Process

Global Data

Local Data

Stack

Stack

PC

PC

Code

Data

Process Data

Kernel Space
Thread Concurrency

Time

T1

T2

T3

CPU

1/26/2005

ComS 425

Spring 2005

18 of 44: Lecture 7
Thread Parallelism

T1

T2

T3

Time
MT codes

• System Calls
  ➢ Only one thread at a time can make a call
    ▪ Commonly called Master (Kernel interacting) Slave
  ➢ Multiple threads can make a call.

• Synchronization
  ➢ Since threads see all the global/local data, it is needed.
  ➢ Mutual Exclusion Lock (Mutex)
    ▪ Similar in spirit to the GUARD/ENDGUARD of Crays.

• Scheduling
  ➢ Done by the Operating System so it “just works”
Value of Threads

- Performance gains from SMP hardware
- Increased Application Throughput
- Increased Application Responsiveness
- Eliminates IPC
- Efficient use of resources
- Ability to make use of concurrency
  - Leads to a more modular program paradigm
- Same executable/OS works on
  - Uniprocessors
  - SMPs
- Single source code for multiple systems/platforms
How do threads improve throughput?

• A program requests services from the OS
  ➢ The program will wait until the service request is complete
  ➢ Leaves CPU idle

• An MT program can do other work while the thread running the service request waits.

• Also easier to synchronize threads no IPC.

• Shared process resources
  ➢ Memory and Virtual memory pages
  ➢ I/O file descriptors and offset
Question?

• **Question:** What can threads do that cannot be done by processes that share memory.
  - Either via native shared memory address space or
  - System V shared memory segments.

• **Answer:** nothing

• **Why threads:**
  - Lower concurrency/parallel overheads.
    - Process creation is 30 times thread creation
    - Synchronization is 10 times faster for threads
    - Context switching among threads is 5 times faster.
Threads Standards

- **OpenMP**
- **POSIX** (Pthreads).
  - Most Unix and even windows.
- **Java Threads.**
- **DCE threads**
  - Dying
- **DEC threads**
  - Should be dead and will be.
- **SOLARIS**
  - dying
- **Win32**
  - Should be dying but it won’t M$ 😊
Thread Scheduling

• In general don’t have to worry about this.
• However there are system specifics that may force you to change the scheduling model.
  ➢ Each vendor’s interpretation of the standard 😊
• Proper systems schedule threads of a process with the same priority as they schedule other processes.
  ➢ Benefit. Your MT program has a bigger crack at the schedule because of the scheduling policy.
The concept of LWP

• **Light weight processes are an abstraction idea that helps to conceptualize the way threads work.**
  - Each thread is an entity of instructions, data, code, etc that can be executed.
    - The pool of available threads have work to do.
    - Each thread is a peer in the context of scheduling.

• **The LWP in essence “connects” a member of a pool of threads to an available member of a processor pool.**

• **Power tool analogy**
  - Limited Battery packs and lots of tools.
  - Multiple battery packs is like multiple processors.
Load Balancing

• The process of dividing work done by each available processor into approximately equal amounts.

• In a multiuser environment, the number of available processors is constantly changing.

• Dynamic load balancing done by creating small granularity parallelism.
  ➢ For example, work is allocated to each task one iteration at a time.
    ▪ This is not always the best 😊
  ➢ A rough example of this is the Unix Time Sharing system.
Load Balancing [2]

• The relationship between load balancing and the extent of parallelism is as follows:
  ➢ The higher the extent of parallelism, the easier it is to balance the workload evenly across the processors.
  ➢ Small granularity parallelism is easier to balance across available processors than large granularity parallelism.
  ➢ Small granularity parallelism generates more overhead than large granularity parallelism.
  ➢ Synchronization is required each time a chunk of work is allocated to a processor.
  ➢ You must evaluate the trade-offs between load balancing and overhead.
Load Balancing [3]

- You can resolve a load imbalance problem by
  - adjusting the size of work for each task,
  - reorganizing a nested loop structure,
  - selecting an intermediate level loop to be parallelized instead of the outer loop.
  - Make sure there is enough work to parallelize
Shared Memory Model

- **Multiple tasks involved**
  - Executing concurrently
- **Can be either the same or different functionality**
  - Functional parallelism
- **Communication is via reading and writing to shared memory**
  - Process based with OS extensions
    - System V Shared memory
  - Thread based
Shared Memory Programming

• **Data and task placement is transparent**
  - Programmer does NOT control this explicitly.
  - Easier to program harder to optimize for performance.

• **Replication and migration of data/processes is transparent**
  - Cache thrashing
  - Processes migrate among processors via the Unix OS time-scheduling algorithm.
  - Also makes performance prediction a challenge.
OpenMP

• **Industry Standard developed in 1997**
  - This is relatively new
  - Few free implementations for C, none for Fortran
    - Most based on Pthreads or Java Threads.
  - Based on common thread based models in existence
    - Solaris, SGI, Cray parallel thread models.
    - Pthreads
    - More structure imposed than a general thread model.
  - Goal to be the standard shared memory programming model for single address space computing.

• **OpenMP Architecture Review Board (ARB)**
  - Runs the show
  - [http://www.openmp.org](http://www.openmp.org)
OpenMP Advantages

• Ability to parallelize small parts of an application at a time.
  ➢ Start with the most time critical ones 😊
• Simple or complex algorithms can be expressed using OpenMP
  ➢ Proportional amounts of work for each
• The size of the code is only marginally increased
  ➢ 5 to 20 percent
• Reasonably clear expression of parallelism so code is “easy” to read.
  ➢ Not many extra calls.
• Integrated debuggers starting to become available.
• Single source code for both sequential and parallel libraries
  ➢ Non OpenMP compilers ignore the OMP directives
    ▪ With proper insertion of OpenMP directives.
OpenMP Programming Model

- The Application Programmer Interface or API is a combination of:
  - Directives
  - Runtime library routines
  - Environment variables
The OpenMP API

• **Breaks down into three categories.**
  - **Expression of parallelism**
    - Controlling the flow of the code
  - **Constructs for sharing data among threads.**
    - Communication mechanisms
  - **Synchronization constructs**
    - Coordination or interaction of threads.
Parallel Control Structures

• Follows the fork/join model.
  ➢ An abstraction that is portable
    ▪ Actual implementation details may be different.

• Only a minimal set of constructs

• Parallel Regions
  ➢ Blocks of code that are executed on every thread.

• Parallel do and Parallel for
  ➢ Divides work in a loop among multiple threads
  ➢ Other similar constructs.
  ➢ Loop-level parallelism.

• Plus others
Communication/Data Environment

- Threads have the shared memory of the process
  - Separate stacks and program counters
- Variables in a program have different scope or execution context.
  - Shared
    - Among all threads
  - Private
    - Each thread has its own copy
  - Reduction
    - Both a private and shared context.
Synchronization

• **Mutual exclusions**
  - Control access to shared variables.
  - Control access to functionality or regions of code.
  - Critical sections

• **Event synchronizations**
  - Barriers
  - Implied barriers (joining of threads)
    - At the end of a for or do loop construct.
General Directive Sentinels

- **Fortran 77**
  - *$omp directive [options]*
  - C$omp directive [options]
  - !$omp directive [options]
- **Fortran 90,95**
  - !$omp directive [options]
- **C/C++**
  - #pragma omp directive [options]
- **Continuation Syntax:**
  - F77:   !$omp directive
            !$omp+    directive
  - C/C++  #pragma omp directive directive

1/26/2005            ComS 425            Spring 2005            40 of 44: Lecture 7
Simple program (C)

```c
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    int tid;
    printf(" Hello world from threads:\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("<%d>\n",tid);
    }
    printf(" I am sequential now\n");
}
```
Output

Hello world from threads:
<0>
<1>
<2>
<3>
<4>

I am sequential now

Is this deterministic?
Output Reality

Hello world from threads:
<0>
<3>
<4>
<2>
<1>

I am sequential now

Order of thread execution is scheduled by the OS. You the Programmer have no control over this and you need to KNOW that!!
Simple Program (F77)

program hello
implicit none
integer tid
integer omp_get_thread_num
external omp_get_thread_num
write(6,*)' Hello world from threads:'
!$omp parallel private(tid)
tid = omp_get_thread_num()
write(6,'(1x,a1,i4,a1)')'<',tid,'>'
!$omp end parallel
write(6,*)' I am sequential now'
end