
Shared Memory Programming OpenMP and Threads

Horst D. Simon

hdsimon@lbl.gov

www.cs.berkeley.edu/~kamil/cs267

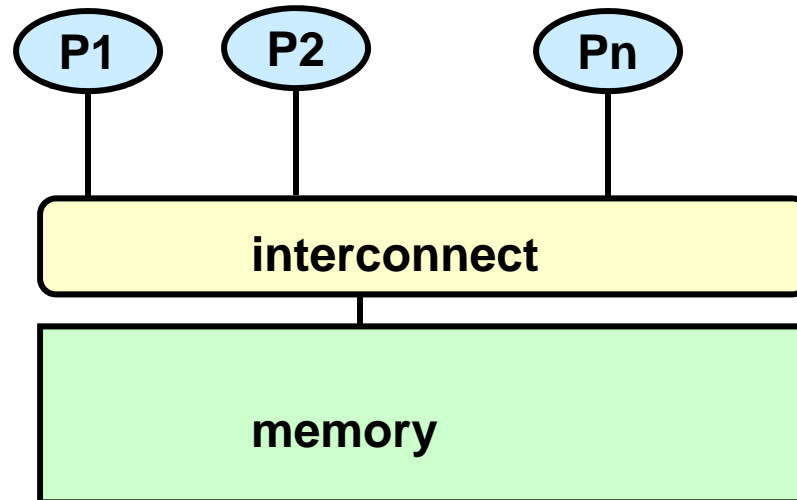
Outline

- Memory consistency: the dark side of shared memory
 - Hardware review and a few more details
 - What this means to shared memory programmers
- Parallel Programming with Threads
- Parallel Programming with OpenMP
 - See <http://www.nersc.gov/nusers/help/tutorials/openmp/>
 - Slides on OpenMP derived from: U.Wisconsin tutorial, which in turn were from LLNL, NERSC, U. Minn, and OpenMP.orgg
- Summary

Shared Memory Hardware and Memory Consistency

Basic Shared Memory Architecture

- Processors all connected to a large shared memory
 - Where are caches?



- Now take a closer look at structure, costs, limits, programming

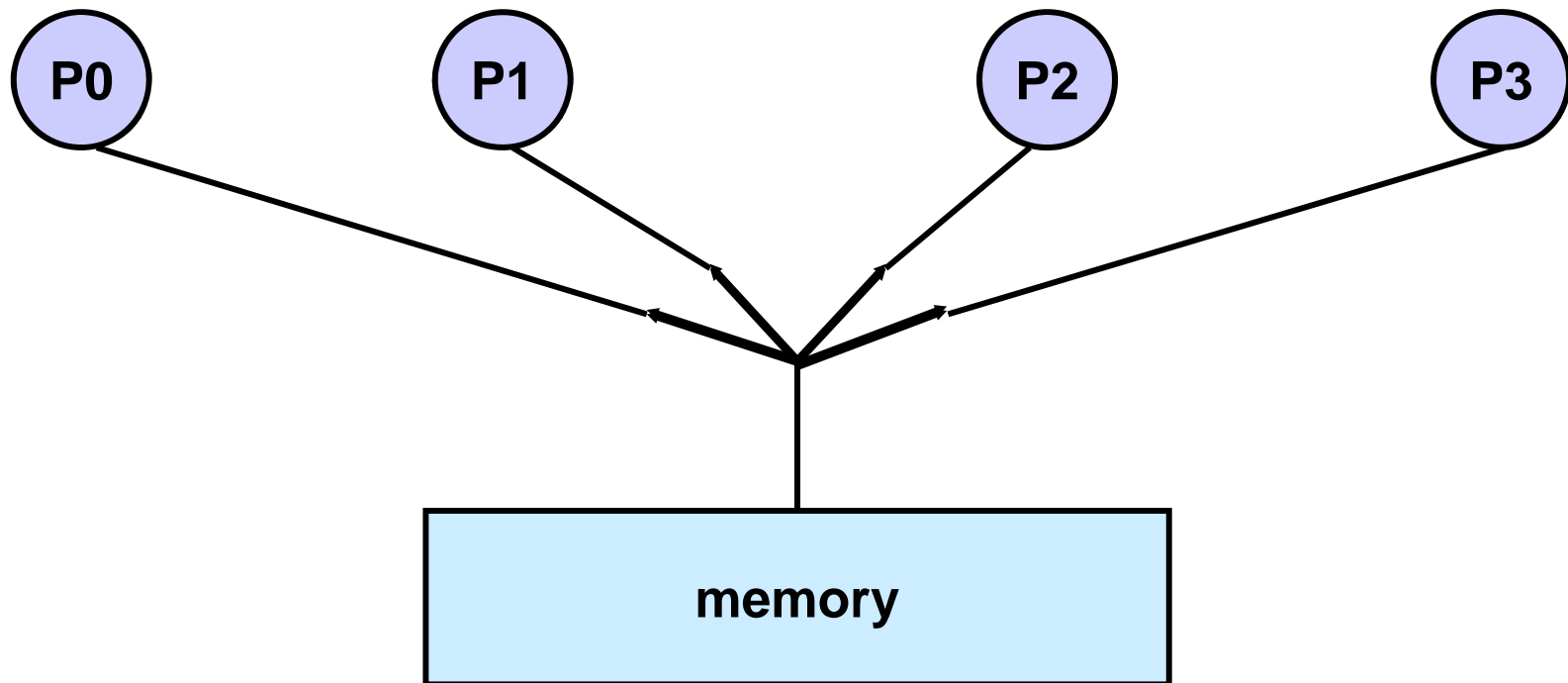
Intuitive Memory Model

- Reading an address should **return the last value written** to that address
- Easy in uniprocessors
 - **except for I/O**
- Cache coherence problem in MPs is more pervasive and more performance critical
- More formally, this is called **sequential consistency**:

“A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” [Lamport, 1979]

Sequential Consistency Intuition

- Sequential consistency says the machine *behaves as if* it does the following

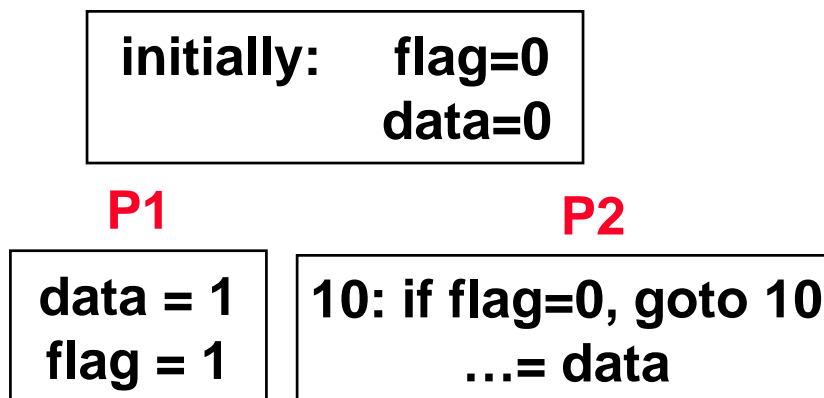


Memory Consistency Semantics

What does this imply about program behavior?

- No process ever sees “garbage” values, i.e., average of 2 values
- Processors always see values written by some processor
- The value seen is constrained by program order on all processors
 - Time always moves forward
- Example: *spin lock*
 - P1 writes data=1, then writes flag=1
 - P2 waits until flag=1, then reads data

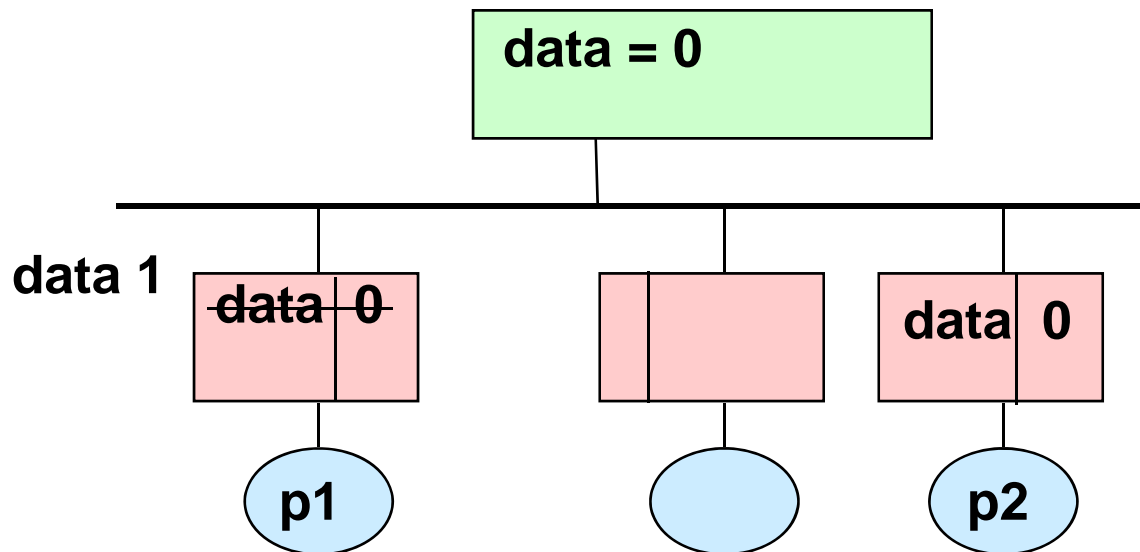
If P2 sees the new value of flag (=1), it must see the new value of data (=1)



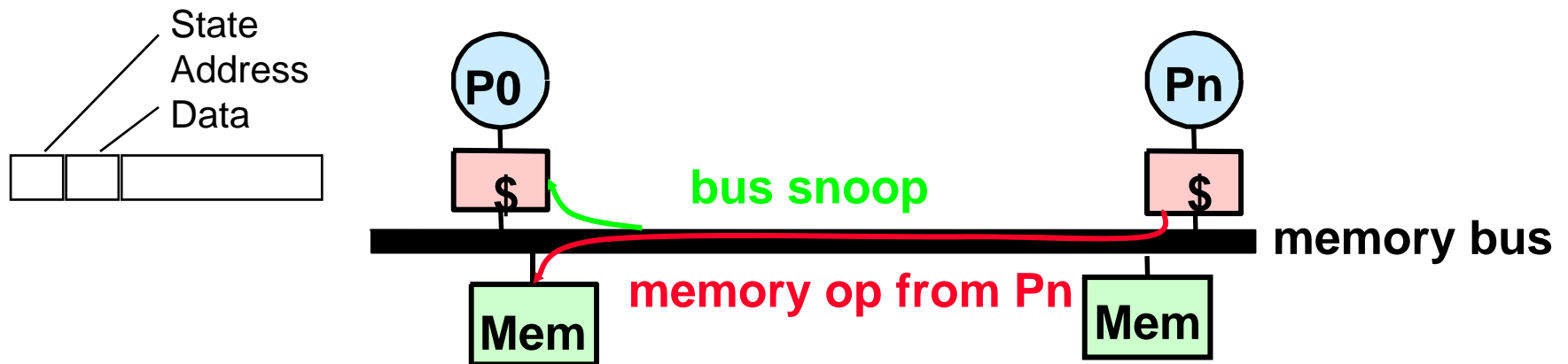
If P2 reads flag	Then P2 may read data
0	1
0	0
1	1

If Caches are Not “Coherent”

- Coherence means different copies of same location have same value, incoherent otherwise:
- p1 and p2 both have cached copies of data (= 0)
- p1 writes data=1
 - May “write through” to memory
- p2 reads data, but gets the “stale” cached copy
 - This may happen even if it read an updated value of another variable, flag, that came from memory

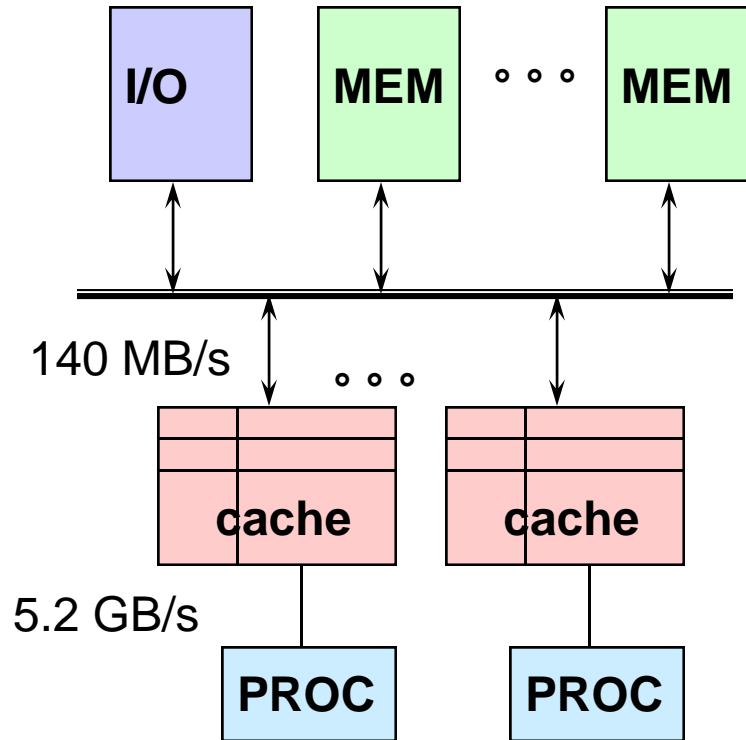


Snoopy Cache-Coherence Protocols



- Memory bus is a broadcast medium
- Caches contain information on which addresses they store
- Cache Controller “snoops” all transactions on the bus
 - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
 - Take action to ensure coherence
 - invalidate, update, or supply value
 - Many possible designs (see CS252 or CS258)

Limits of Bus-Based Shared Memory



Assume:

- 1 GHz processor w/o cache
- => 4 GB/s inst BW per processor (32-bit)
- => 1.2 GB/s data BW at 30% load-store

Suppose 98% inst hit rate and 95% data hit rate

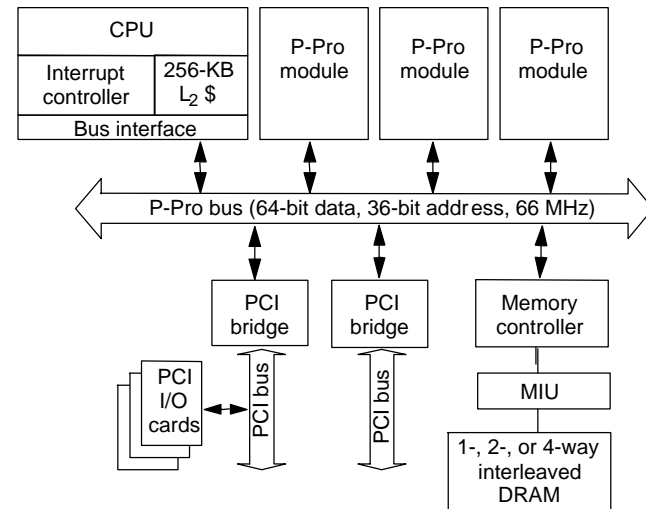
- => 80 MB/s inst BW per processor
- => 60 MB/s data BW per processor
- => 140 MB/s combined BW

Assuming 1 GB/s bus bandwidth
∴ 8 processors will saturate bus

Sample Machines

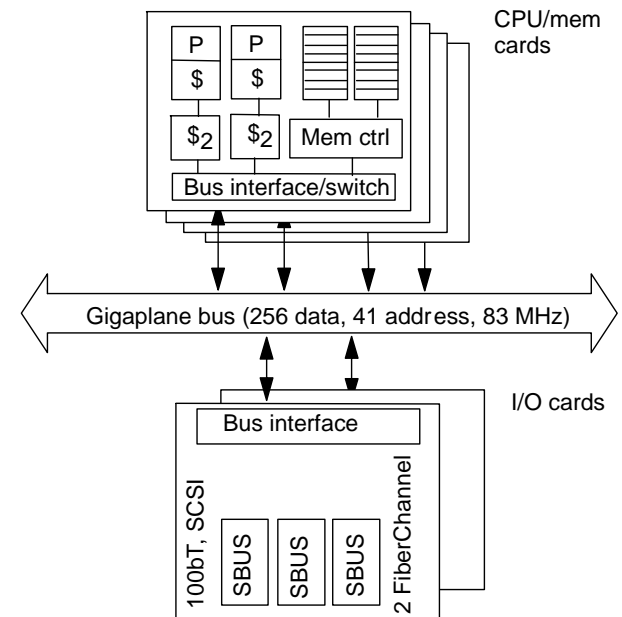
- Intel Pentium Pro Quad

- Coherent
- 4 processors



- Sun Enterprise server

- Coherent
- Up to 16 processor and/or memory-I/O cards



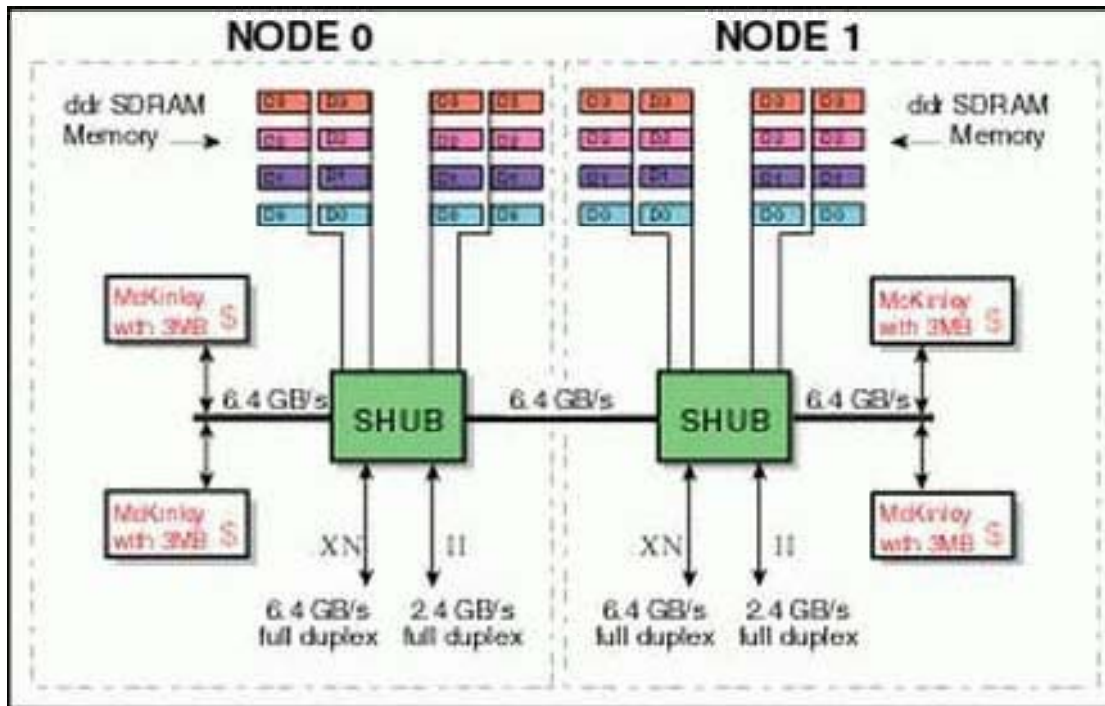
- IBM Blue Gene/L

- L1 not coherent, L2 shared

Basic Choices in Memory/Cache Coherence

- Keep Directory to keep track of which memory stores latest copy of data
- Directory, like cache, may keep information such as:
 - Valid/invalid
 - Dirty (inconsistent with memory)
 - Shared (in another caches)
- When a processor executes a write operation to shared data, basic design choices are:
 - With respect to memory:
 - Write through cache: do the write in memory as well as cache
 - Write back cache: wait and do the write later, when the item is flushed
 - With respect to other cached copies
 - Update: give all other processors the new value
 - Invalidate: all other processors remove from cache
- See CS252 or CS258 for details

SGI Altix 3000



“Best of Show”
-LinuxWorld 2003

- A node contains up to 4 Itanium 2 processors and 32GB of memory
- Network is SGI’s NUMAlink, the NUMAflex interconnect technology.
- Uses a mixture of snoopy and directory-based coherence
- Up to 512 processors that are cache coherent (global address space is possible for larger machines)

Cache Coherence and Sequential Consistency

- There is a lot of hardware/work to ensure coherent caches
 - Never more than 1 version of data for a given address in caches
 - Data is always a value written by some processor
- But other HW/SW features may break sequential consistency (SC):
 - The compiler reorders/removes code (e.g., your spin lock)
 - The compiler allocates a register for flag on Processor 2 and spins on that register value without ever completing
 - Write buffers (place to store writes while waiting to complete)
 - Processors may reorder writes to merge addresses (not FIFO)
 - Write X=1, Y=1, X=2 (second write to X may happen before Y's)
 - Prefetch instructions cause read reordering (read data before flag)
 - The network reorders the two write messages.
 - The write to flag is nearby, whereas data is far away.
 - Some of these can be prevented by declaring variables “volatile”
- Most current commercial SMPs give up SC
 - A correct program on a SC processor may be incorrect on one that is not

Programming with Weaker Memory Models than SC

- Possible to reason about machines with fewer properties, but difficult
- Some rules for programming with these models
 - Avoid race conditions
 - Use system-provided synchronization primitives
 - If you have race conditions on variables, make them volatile
 - At the assembly level, may use “fences” (or analogs) directly
- The high level language support for these differs
 - Built-in synchronization primitives normally include the necessary fence operations
 - lock (), ... only one thread at a time allowed here.... unlock()
 - Region between lock/unlock called **critical region**
 - For performance, need to keep critical region short

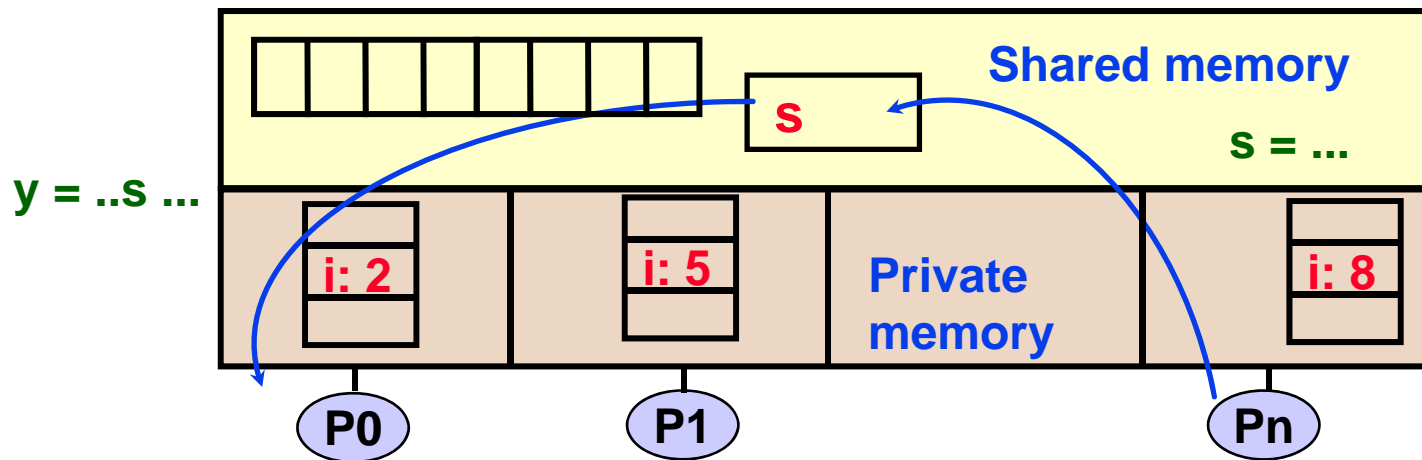
Sharing: A Performance Problem

- True sharing
 - Frequent writes to a variable can create a bottleneck
 - OK for read-only or infrequently written data
 - Technique: make copies of the value, one per processor, if this is possible in the algorithm
 - Example problem: the data structure that stores the freelist/heap for malloc/free
- False sharing
 - Cache block may also introduce artifacts
 - Two distinct variables in the same cache block
 - Technique: allocate data used by each processor contiguously, or at least avoid interleaving in memory
 - Example problem: an array of ints, one written frequently by each processor (many ints per cache line)

Parallel Programming with Threads

Programming Model 1: Shared Memory

- Program is a collection of threads of control.
 - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
 - Threads communicate **implicitly** by writing and reading shared variables.
 - Threads coordinate by **synchronizing** on shared variables



Shared Memory Programming

Several Thread Libraries

- PTHREADS is the POSIX Standard
 - Solaris threads are very similar
 - Relatively low level
 - Portable but possibly slow
- OpenMP is newer standard
 - Support for scientific programming on shared memory
 - <http://www.openMP.org>
- P4 (Parmacs) is an older portable package
 - Higher level than Pthreads
 - <http://www.netlib.org/p4/index.html>

Common Notions of Thread Creation

- cobegin/coend

```
cobegin
    job1(a1);
    job2(a2);
coend
```

- Statements in block may run in parallel
- cobegins may be nested
- Scoped, so you cannot have a missing coend

- fork/join

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

- Forked procedure runs in parallel
- Wait at join point if it's not finished

- future

```
v = future(job1(a1));
... = ...v...;
```

- Future expression evaluated in parallel
- Attempt to use return value will wait

- Cobegin cleaner than fork, but fork is more general
- Futures require some compiler (and likely hardware) support

Overview of POSIX Threads

- POSIX: *Portable Operating System Interface for UNIX*
 - Interface to Operating System utilities
- PThreads: The POSIX threading interface
 - System calls to create and synchronize threads
 - Should be relatively uniform across UNIX-like OS platforms
- PThreads contain support for
 - Creating parallelism
 - Synchronizing
 - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *,
                  const pthread_attr_t *,
                  void * (*)(void *),
                  void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                        &thread_fun; &fun_arg);
```

- `thread_id` is the thread id or handle (used to halt, etc.)
- `thread_attribute` various attributes
 - standard default values obtained by passing a NULL pointer
- `thread_fun` the function to be run (takes and returns void*)
- `fun_arg` an argument can be passed to `thread_fun` when it starts
- `errorcode` will be set nonzero if the create operation fails

Simple Threading Example

```
void* SayHello(void *foo) {  
    printf( "Hello, world!\n" );  
    return NULL;  
}
```

Compile using gcc -lpthread
See Millennium/NERSC docs for
paths/modules

```
int main() {  
    pthread_t threads[16];  
    int tn;  
    for(tn=0; tn<16; tn++) {  
        pthread_create(&threads[tn], NULL, SayHello, NULL);  
    }  
    for(tn=0; tn<16 ; tn++) {  
        pthread_join(threads[tn], NULL);  
    }  
    return 0;  
}
```

Loop Level Parallelism

- Many scientific application have parallelism in loops

- With threads:

```
... my_stuff [n][n];  
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        ... pthread_create (update_cell, ...,  
                             my_stuff[i][j]);
```

Also need i & j



- But overhead of thread creation is nontrivial
 - update_cell should have a significant amount of work
 - 1/pth if possible

Shared Data and Threads

- Variables declared outside of main are shared
- Object allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems
- Often done by creating a large “thread data” struct
 - Passed into all threads as argument
 - Simple example:

```
char *message = "Hello World!\n";

pthread_create( &thread1,
               NULL,
               (void*)&print_fun,
               (void*) message);
```

Setting Attribute Values

- Once an initialized attribute object exists, changes can be made. For example:
 - To change the stack size for a thread to 8192 (before calling `pthread_create`), do this:
 - `pthread_attr_setstacksize(&my_attributes, (size_t)8192);`
 - To get the stack size, do this:
 - `size_t my_stack_size;`
`pthread_attr_getstacksize(&my_attributes, &my_stack_size);`
- Other attributes:
 - Detached state – set if no other thread will use `pthread_join` to wait for this thread (improves efficiency)
 - Guard size – use to protect against stack overflow
 - Inherit scheduling attributes (from creating thread) – or not
 - Scheduling parameter(s) – in particular, thread priority
 - Scheduling policy – FIFO or Round Robin
 - Contention scope – with what threads does this thread compete for a CPU
 - Stack address – explicitly dictate where the stack is located
 - Lazy stack allocation – allocate on demand (lazy) or all at once, “up front”

Recall Data Race Example from Last Time

```
static int s = 0;
```

Thread 1

```
for i = 0, n/2-1  
  s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1  
  s = s + f(A[i])
```

- Problem is a race condition on variable `s` in the program
- A **race condition** or **data race** occurs when:
 - two processors (or two threads) access the same variable, and at least one does a write.
 - The accesses are concurrent (not synchronized) so they could happen simultaneously

Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
 - SPMD “Single Program Multiple Data”
- simple use of barriers -- all threads hit the same one

```
work_on_my_subgrid();  
barrier;  
read_neighboring_values();  
barrier;
```

- more complicated -- barriers on branches (or loops)

```
if (tid % 2 == 0) {  
    work1();  
    barrier  
} else { barrier }
```

- barriers are not provided in all thread libraries

Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;  
pthread_barrier_init(&b, NULL, 3);
```

- The second argument specifies an object attribute; using NULL yields the default attributes.
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

- This barrier could have been statically initialized by assigning an initial value created using the macro

```
PTHREAD_BARRIER_INITIALIZER(3).
```

Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

- threads are working mostly independently
- need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
access data
release(l);
```

- Java and other languages have lexically scoped synchronization
 - similar to cobegin/coend vs. fork and join tradeoff
- Semaphores give guarantees on “fairness” in getting the lock, but the same idea of mutual exclusion
- Locks only affect processors using them:
 - pair-wise synchronization

Mutexes in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Multiple mutexes may be held, but can lead to deadlock:

thread1	thread2
lock(a)	lock(b)
lock(b)	lock(a)

Introduction to OpenMP

- What is OpenMP?
 - Open specification for Multi-Processing
 - “Standard” API for defining multi-threaded shared-memory programs
 - www.openmp.org – Talks, examples, forums, etc.
- High-level API
 - Preprocessor (compiler) directives (~ 80%)
 - Library Calls (~ 19%)
 - Environment Variables (~ 1%)

Summary of Programming with Threads

- POSIX Threads are based on OS features
 - Can be used from multiple languages (need appropriate header)
 - Familiar language for most of program
 - Ability to shared data is convenient
- Pitfalls
 - Data race bugs are very nasty to find because they can be intermittent
 - Deadlocks are usually easier, but can also be intermittent
- Researchers look at transactional memory an alternative
- OpenMP is commonly used today as an alternative

Parallel Programming in OpenMP

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with “light” syntax
 - Exact behavior depends on OpenMP *implementation!*
 - Requires compiler support (C or Fortran)
- OpenMP will:
 - Allow a programmer to separate a program into *serial regions* and *parallel regions*, rather than T concurrently-executing threads.
 - Hide stack management
 - Provide synchronization constructs
- OpenMP will not:
 - Parallelize automatically
 - Guarantee speedup
 - Provide freedom from data races

Motivation

- Thread libraries are hard to use
 - P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
 - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness
- Wouldn't it be nice to write serial programs and somehow parallelize them "automatically"?
 - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence
 - It is not automatic: you can still make errors in your annotations

Motivation – OpenMP

```
int main() {  
  
    // Do this part in parallel  
  
    printf( "Hello, World!\n" );  
  
    return 0;  
}
```

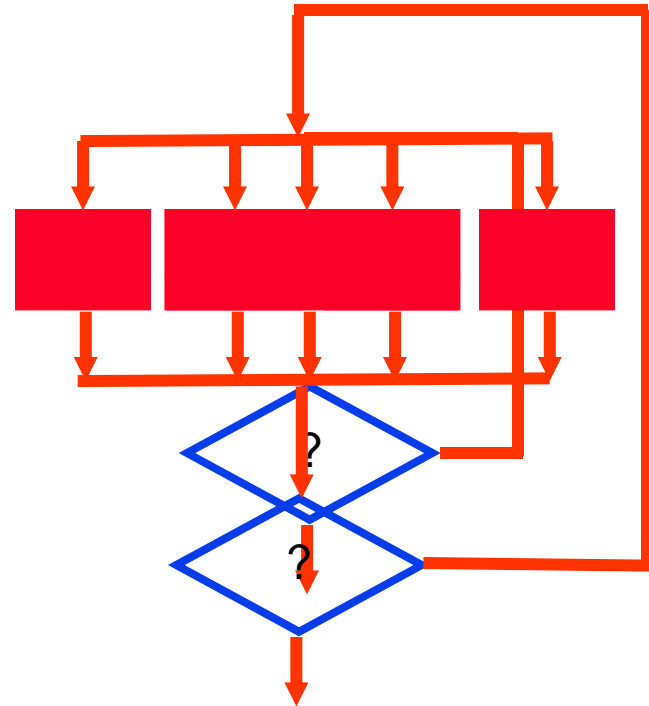
Motivation – OpenMP

```
int main() {  
  
    omp_set_num_threads(16);  
  
    // Do this part in parallel  
    #pragma omp parallel  
    {  
        printf( "Hello, World!\n" );  
    }  
  
    return 0;  
}
```

Programming Model – Concurrent Loops

- OpenMP easily parallelizes loops
 - Requires: No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds for each thread directly from *serial* source

```
#pragma omp parallel for  
  
for( i=0; i < 25; i++ )  
{  
    printf( "Foo" );  
}
```



Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team
 - `static([chunk])` divides iterations statically between threads
 - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
 - Default `[chunk]` is `ceil(# iterations / # threads)`
 - `dynamic([chunk])` allocates `[chunk]` iterations per thread, allocating an additional `[chunk]` iterations when a thread finishes
 - Forms a logical work queue, consisting of all loop iterations
 - Default `[chunk]` is 1
 - `guided([chunk])` allocates dynamically, but `[chunk]` is exponentially reduced with each allocation

Programming Model – Data Sharing

- Parallel programs often employ two types of data
 - Shared data, visible to all threads, similarly named
 - Private data, visible to a single thread (often stack-allocated)
- PThreads:
 - Global-scoped variables are shared
 - Stack-allocated variables are private
- OpenMP:
 - `shared` variables are shared
 - `private` variables are private

```
// shared, globals
int bigdata[1024];

void* foo(void* bar) {
    int private, stack;
    int tid;
    #pragma omp parallel \
    / shared (bigdata) \
    private*( tid )
} {
    /* Calc. here */
}
}
```

Programming Model - Synchronization

- OpenMP Synchronization

- OpenMP Critical Sections

- Named or unnamed
 - No *explicit* locks

```
#pragma omp critical
{
    /* Critical code here */
}
```

- Barrier directives

```
#pragma omp barrier
```

- Explicit Lock functions

- When all else fails – may require `flush` directive

```
omp_set_lock( lock 1 );
/* Code goes here */
omp_unset_lock( lock 1 );
```

- Single-thread regions *within* parallel regions

- `master`, `single` directives

```
#pragma omp single
{
    /* Only executed once */
}
```

Microbenchmark: Grid Relaxation

```
for( t=0; t < t_steps; t++) {
    #pragma omp parallel for \
        shared(grid,x_dim,y_dim) private(x,y)

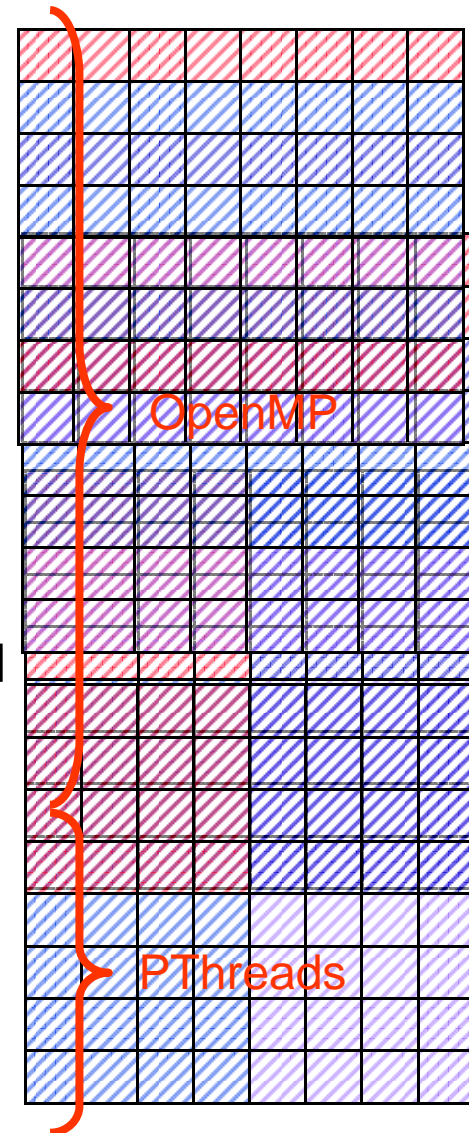
    for( x=0; x < x_dim; x++) {
        for( y=0; y < y_dim; y++) {
            grid[x][y] = /* avg of neighbors */
        }
    }

    // Implicit Barrier Synchronization

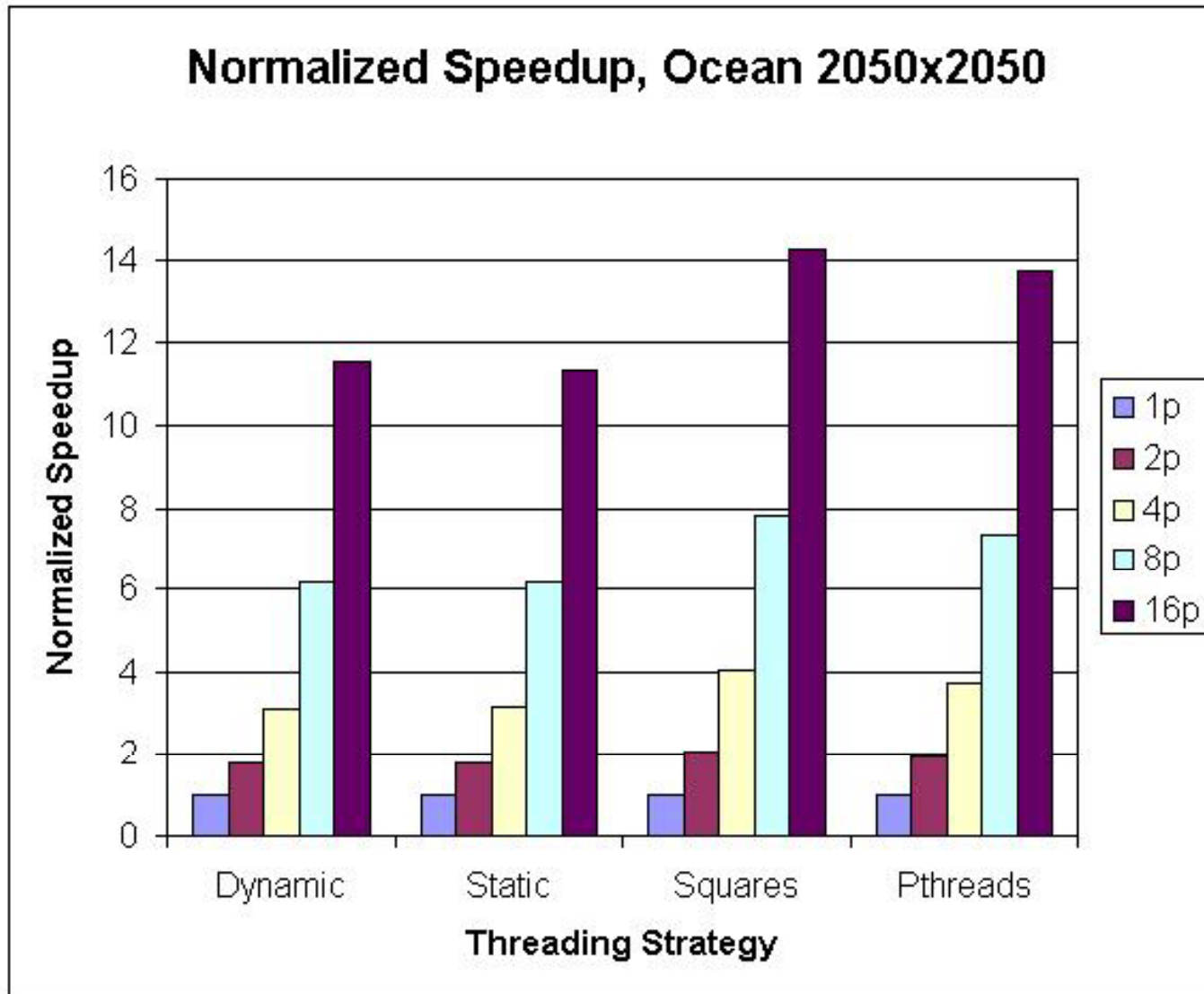
    temp_grid = grid;
} grid = other_grid;
other_grid = temp_grid;
```

Microbenchmark: Structured Grid

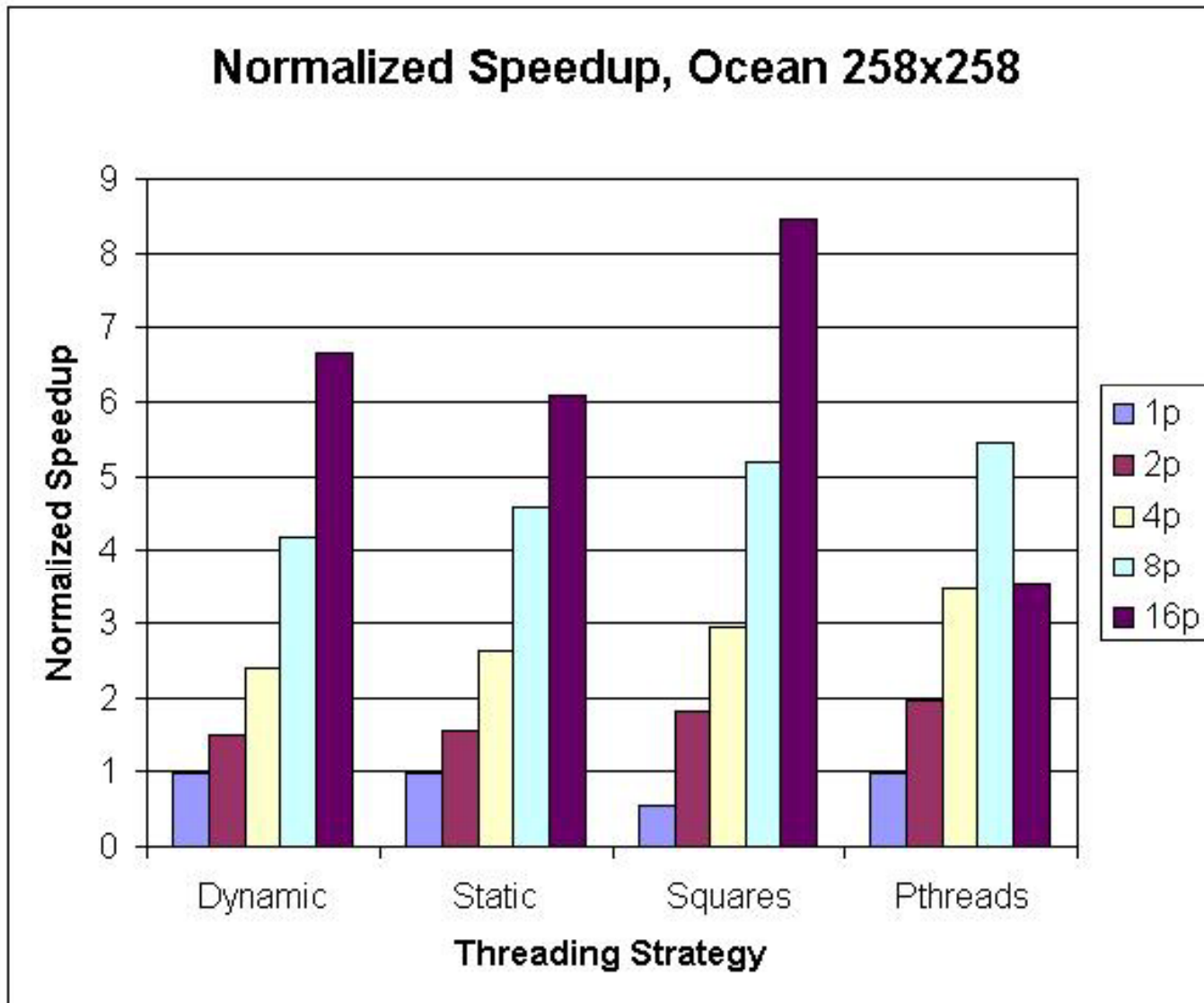
- **ocean_dynamic** – Traverses entire ocean, row-by-row, assigning row iterations to threads with `dynamic` scheduling.
- **ocean_static** – Traverses entire ocean, row-by-row, assigning row iterations to threads with `static` scheduling.
- **ocean_squares** – Each thread traverses a square-shaped section of the ocean. Loop-level scheduling not used—loop bounds for each thread are determined explicitly.
- **ocean_pthreads** – Each thread traverses a square-shaped section of the ocean. Loop bounds for each thread are determined explicitly.



Microbenchmark: Ocean



Microbenchmark: Ocean



Microbenchmark: GeneticTSP

- Genetic heuristic-search algorithm for approximating a solution to the traveling salesperson problem
- Operates on a *population* of possible TSP paths
 - Forms new paths by combining known, good paths (*crossover*)
 - Occasionally introduces new random elements (*mutation*)
- Variables:
 - N_p – Population size, determines search space and working set size
 - N_g – Number of generations, controls effort spent refining solutions
 - r_C – Rate of crossover, determines how many new solutions are produced and evaluated in a generation
 - r_M – Rate of mutation, determines how often new (random) solutions are introduced

Microbenchmark: GeneticTSP

```
while( current_gen < Ng ) {  
Breed rC*Np new solutions:  
    Select two parents  
    Perform crossover()  
    Mutate() with probability rM  
    Evaluate() new solution  
  
    Identify least-fit rC*Np solutions:  
    Remove unfit solutions from population  
  
    current_gen++  
}  
  
return the most fit solution found
```

Outer loop has data dependencies between iterations, as the population is not a loop invariant. `crossover()`, and `evaluate()` have varying runtimes. `evaluate()` can find least-fit population members in parallel, but only one thread should actually delete solutions.

Microbenchmark: GeneticTSP

- **dynamic_tsp** – Parallelizes both breeding loop and survival loop with OpenMP's `dynamic` scheduling
- **static_tsp** – Parallelizes both breeding loop and survival loop with OpenMP's `static` scheduling
- **tuned_tsp** – Attempt to tune scheduling. Uses `guided` (exponential allocation) scheduling on breeding loop, `static` predicated scheduling on survival loop.
- **pthread_tsp** – Divides iterations of breeding loop evenly among threads, conditionally executes survival loop in parallel

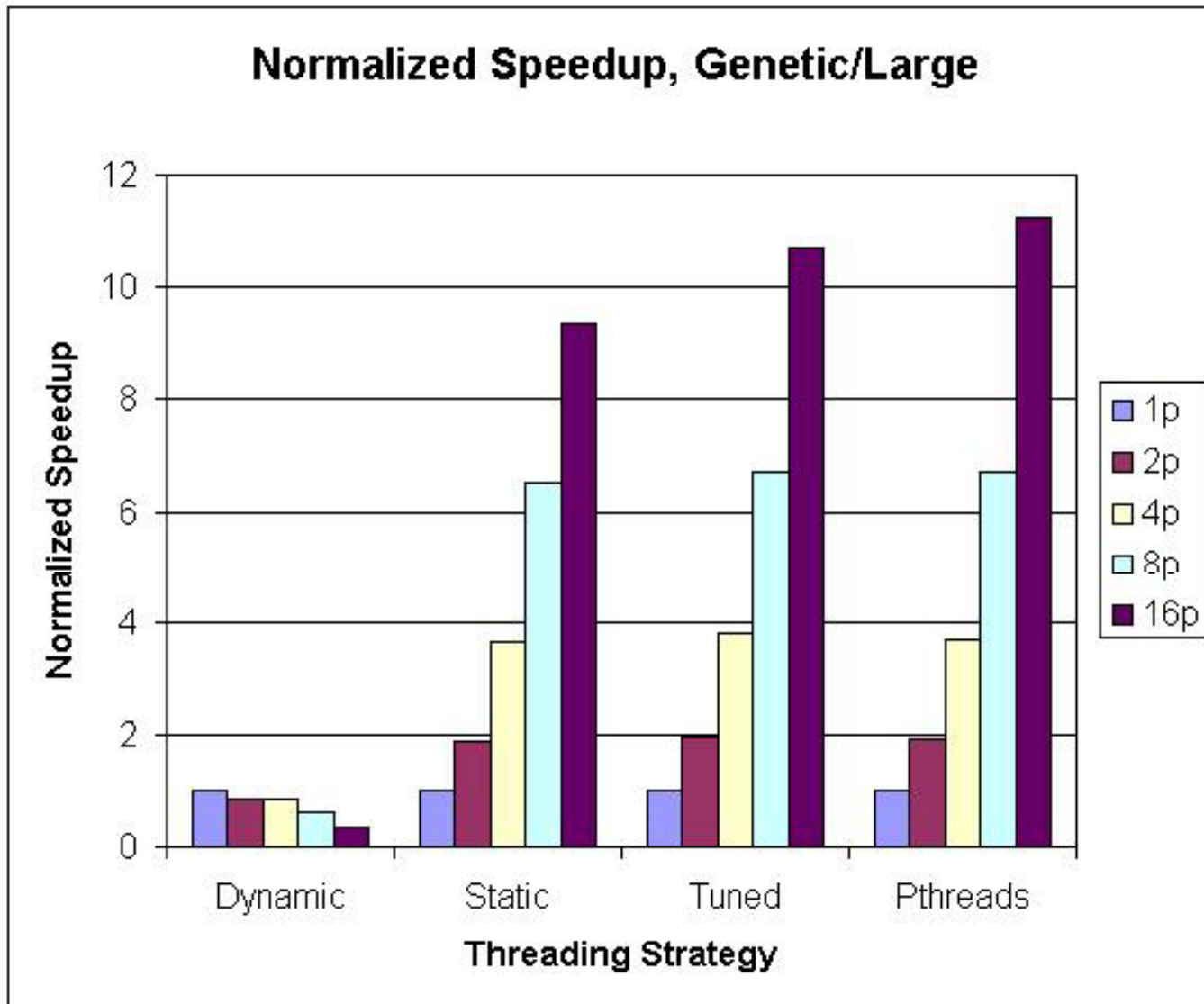


OpenMP



PThreads

Microbenchmark: GeneticTSP



Evaluation

- OpenMP scales to 16-processor systems
 - Was overhead too high?
 - In some cases, yes
 - Did compiler-generated code compare to hand-written code?
 - Yes!
 - How did the loop scheduling options affect performance?
 - `dynamic` or `guided` scheduling helps loops with variable iteration runtimes
 - `static` or predicated scheduling more appropriate for shorter loops
- Is OpenMP the right tool to parallelize scientific application?

SpecOMP (2001)

- Parallel form of SPEC FP 2000 using Open MP, larger working sets
 - Aslot et. Al., Workshop on OpenMP Apps. and Tools (2001)
- Many of CFP2000 were “straightforward” to parallelize:
 - *ammp*: 16 Calls to OpenMP API, 13 #pragmas, converted linked lists to vector lists
 - *applu*: 50 directives, mostly `parallel` or `do`
 - *fma3d*: 127 lines of OpenMP directives (60k lines total)
 - *mgrid*: automatic translation to OpenMP
 - *swim*: 8 loops parallelized

OpenMP Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
 - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
 - Scalable
 - Portable
- Not a silver bullet for all applications

More Information

- www.openmp.org
 - OpenMP official site
- www.llnl.gov/computing/tutorials/openMP/
 - A handy OpenMP tutorial
- www.nersc.gov/nusers/help/tutorials/openmp/
 - Another OpenMP tutorial and reference

What to Take Away?

- Programming shared memory machines
 - May allocate data in large shared region without too many worries about where
 - Memory hierarchy is critical to performance
 - Even more so than on uniprocessors, due to coherence traffic
 - For performance tuning, watch sharing (both true and false)
- Semantics
 - Need to lock access to shared variable for read-modify-write
 - Sequential consistency is the natural semantics
 - Architects worked hard to make this work
 - Caches are coherent with buses or directories
 - No caching of remote data on shared address space machines
 - But compiler and processor may still get in the way
 - Non-blocking writes, read prefetching, code motion...
 - Avoid races or use machine-specific fences carefully