Mar 12 — Parallelism and Shared Memory Hierarchy I

Rutgers University
Review: Classical Three-pass Compiler

Parallelism Optimization
- Can happen at any stage
- Front end: language design for expressing parallelism
- Middle end: dependence analysis and automatic parallelization/vectorization
- Back end: instruction level parallelism

Parallelization efficiency is highly correlated with shared memory locality

So is shared memory locality enhancement!
Classical Example: Matrix Multiplication

**Basic Algorithm**

```plaintext
for (i=0; i< n; i++)
    for (j=0; j< n; j++){
        Z[i,j] = 0.0;
        for (k=0; k< n; k++)
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
    }
```

Computation-intensive: 3n² locations, n³ operations.
The calculations of each element of Z are independent.
Matrix Multiplication — Serial Execution

Assume $c$ elements per cache line ($c < n$), and one row can fit into the cache but not the whole matrix

Cache misses by $Z$ are negligible.
Cache misses by $X$ are $\frac{n^2}{c}$.
Cache misses by $Y$ are $n^2 \times n = n^3$.

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    Z[i,j] = 0.0;
  for (k=0; k<n; k++)
    Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
}
Matrix Multiplication — Row-by-Row Parallelization

• $p$: the number of processors.
• Let each processor compute $n/p$ consecutive rows of $Z$.
  • Needs to access $n/p$ rows of $X$ and $Z$, and the entire $Y$.
  • Computes $n^2/p$ elements of $Z$ with $n^3/p$ operations performed.
• The total number of cache lines to the caches of all processors is at least $n^2/c + p*n^2/c$
  • As $p$ increases, the amount of computation decreases, but the communication cost increases.
• A row or column might loaded into multiple processors

```c
for (i=0; i< n; i++)
    for (j=0; j< n; j++){
        Z[i, j] = 0.0;
        for (k=0; k< n; k++)
            Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
    }
```
Matrix Multiplication — Optimizations

- What are the data reuses in matrix multiply?
- Why do the reuses on X and Y result in different cache misses? (In both the serial and parallel runs.)
- A reuse yields locality (or a cache hit) only if
  - the reuse happens soon enough, and
  - the data is reused by the same processor.
Matrix Multiplication — Ways to Improve

c elements per cache line.

\[
\begin{align*}
&\text{for (i=0; i<n; i++)} \\
&\quad \text{for (j=0; j<n; j++)} \{ \\
&\quad\quad \text{Z[i, j] = 0.0;} \\
&\quad\quad \text{for (k=0; k<n; k++)} \\
&\quad\quad\quad \text{Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];} \\
&\quad\} \\
\end{align*}
\]

- Blocking
- Changing data layout

Interchange i and j loops?
Optimization I — Blocking

- One way is to reorder iterations in a loop.
- Basic idea
  - Divide the matrix into submatrices (blocks)
  - Order operations so an entire block is re-used over a short period of time.
Blocking

\[
\begin{align*}
\text{for (ii=0; ii<n; ii+=B)} \\
\text{for (jj=0; jj<n; jj+=B)} \\
\text{for (kk=0; kk<n; kk+=B)} \\
\text{for (i=ii; i<i+i+B; i++)} \\
\text{for (j=jj; j<j+j+B; j++)} \\
\text{for (k=kk; k<k+k+B; k++)} \\
Z[i,j] = Z[i,j] + X[i,k]Y[k,j];
\end{align*}
\]

cache misses:
- X: \(B^3/c\) per block, totally \(n^3/B^3\) blocks to bring,
  hence, \(n^3/Bc\) misses in total.
- Y: \(n^3/Bc\) misses.
- \(2n^3/Bc\) misses together.
• When entire matrices fit in cache
  • Original code: $O(n^2/c)$ misses.
  • Opt code: $2n^3/Bc$.
  • Comparable when $B = n$.
• When one column of $Y$ stays in cache but the entire matrix does not
  • Original code: $O(n^3/c)$ misses.
  • Opt code: $2n^3/Bc$.
• When a column cannot stay in cache
  • Original code: $O(n^3)$ misses.
  • Opt code: $2n^3/Bc$.
• $B$ can be reasonably large: e.g. 200 for a 1MB cache.
• 3X speedup on a uniprocessor.
Blocking — Broader Application

- Applicable to each level of the memory hierarchy.
  - Disk, SSD, Memory, Cache, Registers
  - The idea is the same

- Amenable to parallelization: parallel processors or computer nodes
  - Communication reduced.
  - Linear speedup.
Opt Option 2 — Data Layout Transformation

- Storing $Y$ in column-major order
  - Improves spatial locality
- Some side effects are possible
  - If $Y$ is used in other parts of the program.
  - Need to transform the layout back
Principle in Locality Enhancement

- Two fundamental types of approaches you have seen in previous examples
  - Computation reordering
    - blocking

\[
\text{for } (i=0; i<n; i++)
\text{ for } (j=0; j<n; j++)
\text{ for } (k=0; k<n; k++)
\text{ for } (i=ii; i<ii+B; i++)
\text{ for } (j=jj; j<jj+B; j++)
\text{ for } (k=kk; k<kk+B; k++)
\]
\[
\text{Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];}
\]

- Data layout transformation

- Hybrid: a combination of both
- Both of them can be realized in different ways:
  - polyhedral analysis, graph vertex partition based approach, data packing and etc.
  - static time loop transformation or dynamic runtime transformation
20 min break ...
Parallel Programming Models
Parallel Programming Models

• **Shared Memory Programming**
  - Start a single process and fork threads.
  - Threads carry out work.
  - Threads communicate through shared memory.
  - Threads coordinate through synchronization (also through shared memory).

• **Distributed Memory Programming**
  - Start multiple processes on multiple systems.
  - Processes carry out work.
  - Processes communicate through message-passing.
  - Processes coordinate either through message-passing or synchronization (generates messages).
Non-uniform Memory Access (NUMA) Multicore System

A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.
Cache Coherence

- When the value of a cache line changes, all its copies must either be invalidated or updated.
Snooping for Coherence

- The processors/cores share a bus
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus
Directory based Coherence

• Use a data structure called a directory that stores the status of each cache line

• When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable’s cache line in their caches are invalidated
Shared Memory Programming

- **Threads Model**
  - A thread is a single stream of control in the flow of a program

```python
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product( get_row(a, row),
                        get_col(b, col));
```

- **Taxonomy**
  - Dynamic threads
    - Master thread waits for work, forks new threads, and when threads are done, they terminate
    - Efficient use of resources, but thread creation and termination is time consuming
  - Static threads
    - Pool of threads created and are allocated work, but do not terminate until cleanup.
    - Better performance, but potential waste of system resources.
Programming with Threads

- **PThreads — Portable Operating System Interface [for Unix] (POSIX)**
  - Relatively low level
    - Programmer expresses thread management and coordination
    - Programmer decomposes parallelism and manages schedule
  - Most widely used for systems-oriented code, and also used for some applications

- **OpenMP**
  - Higher-level support for scientific programming on shared memory architectures
    - Programmers identifies parallelism and data properties, and guides scheduling at a high level
    - System decomposes parallelism and manages schedule
  - Arose from a variety of architecture-specific pragmas
Pthread — Thread Creation and Termination

- Pthreads provides two basic functions for specifying thread lifetime in a program:

```c
#include <pthread.h>
int pthread_create (
    pthread_t *thread_handle, const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);
int pthread_join ( 
    pthread_t thread,
    void **ptr);
```

- The function pthread_create forks a thread to run function thread_function.
Pthread — Shared Data and Threads

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

```c
char *message = "Hello World!\n";
pthread_create( &thread1,
    NULL,
    (void*) &print_fun,
    (void*) message);
```
Pthreads — Synchronization with Mutex

```c
if (my_cost < best_cost)
    best_cost = my_cost;
```

• best_cost=100; a global variable
• my_cost is thread local
  • T1: 50
  • T2: 75

• Could produce inconsistent results
  • i.e., if T1 is executed in between T2’s condition check and assignment statement, the result will be 75
Pthreads — Synchronization with Mutex

- *Pthread mechanism to implement mutual exclusion*
  - *Locking is an atomic operation*

- **Syntax**

```c
int
pthread_mutex_init (  
    pthread_mutex_t *mutex_lock,  
    const pthread_mutexattr_t *lock_attr);

int
pthread_mutex_lock (  
    pthread_mutex_t *mutex_lock);

int
pthread_mutex_unlock (  
    pthread_mutex_t *mutex_lock);
```
• If the mutex-lock is already locked, the calling thread blocks
  • busy-waiting (keeps polling on the lock)

• Multiple threads may be blocked at one point of time for one lock
  • when the lock becomes unlocked, scheduler determines which thread gets the lock
Pthreads Mutex

• Example 1: find min

```c
#include <pthread.h>
void *find_min(void *list_ptr);
pthread_mutex_t minimum_value_lock;
int minimum_value, partial_list_size;

main() {
    /* declare and initialize data structures and list */
    minimum_value = MIN_INT;
    pthread_init();
    pthread_mutex_init(&minimum_value_lock, NULL);

    /* initialize lists, list_ptr, and partial_list_size */
    /* create and join threads here */
}
```
void *find_min(void *list_ptr) {
    int *partial_list_pointer, my_min, i;
    my_min = MIN_INT;
    partial_list_pointer = (int *) list_ptr;
    for (i = 0; i < partial_list_size; i++)
        if (partial_list_pointer[i] < my_min)
            my_min = partial_list_pointer[i];
    /* lock the mutex associated with minimum_value and
    update the variable as required */
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
        minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
    pthread_exit(0);
}
Example 2: producer-consumer

- An queue (holding at most one task)
- producer threads enqueue; consumer threads dequeue.

```c
pthread_mutex_t task_queue_lock;
int task_available;

/* other shared data structures here */

main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_mutex_init(&task_queue_lock, NULL);
    /* create and join producer and consumer threads */
}
```
void *producer(void *producer_thread_data) {
    int inserted;
    struct task my_task;
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
• In `mutex_lock` (or `mutex_trylock`), the thread always occupies the CPU.

• Alternative
  
  • Put the thread to sleep and wake it when necessary.
  
  • Use POSIX condition variables
Pthreads — Synchronization with Condition Variables

**Definition**: A data object that allows a thread to block itself until specified data reaches a predefined state.

**The corresponding boolean condition is referred to as a predicate.**

```c
int pthread_cond_wait(pthread_cond_t *cond, 
                      pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
```

```c
int pthread_cond_init(pthread_cond_t *cond, 
                      const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```
Pthreads — Synchronization with Condition Variables

- A call to `pthread_cond_wait`
  - blocks the calling thread
  - releases the associated lock

- A blocked thread is waked up when
  - it receives a signal from another thread or
  - it is interrupted by an OS signal

- Upon being waked up, the thread
  - waits to reacquire the lock and then
  - resumes execution
Example: producer-consumer model again

```c
1   pthread_cond_t cond_queue_empty, cond_queue_full;
2   pthread_mutex_t task_queue_cond_lock;
3   int task_available;
4
5   /* other data structures here */
6
7   main() {
8       /* declarations and initializations */
9       task_available = 0;
10      pthread_init();
11      pthread_cond_init(&cond_queue_empty, NULL);
12      pthread_cond_init(&cond_queue_full, NULL);
13      pthread_mutex_init(&task_queue_cond_lock, NULL);
14      /* create and join producer and consumer threads */
15   }
16```
Why not an “if” for the “while” statement?
Caveat: try not to use a single condition variable for multiple predicates
Pthreads — Synchronization with Barrier

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

- This barrier could have been statically initialized by assigning an initial value created using the macro
  ```c
  PTHREAD_BARRIER_INITIALIZER(3).
  ```