Lab 5: Parallelism

Hung-Wei Tseng
Parallelism in modern computers

• Instruction-level parallelism
  • The ability to execute multiple instructions concurrently

• Thread-level parallelism
  • The ability to execute multiple program instances concurrently

• Data-level parallelism
  • The ability to process data concurrently
Processing models

• SISD — single instruction stream, single data
  • Pipelining instructions within a single program
  • Superscalar
• SIMD — single instruction stream, multiple data
  • Vector instructions
  • GPUs
• MIMD — multiple instruction stream (e.g. multiple threads, multiple processes), multiple data
  • Multicore processors
  • Multiple processors
  • Simultaneous multithreading
Instruction-level parallelism
ILP
We’ve learned quite a lot from the lectures

- Compiler optimizations
- Branch predictions
- Dynamic instruction scheduling
Through data flow graph analysis

1. movl (%rdi), %ecx
2. addq $4, %rdi
3. addl %ecx, %eax
4. cmpq %rdx, %rdi
5. jne .L3
6. movl (%rdi), %ecx
7. addq $4, %rdi
8. addl %ecx, %eax
9. cmpq %rdx, %rdi
10. jne .L3
11. movl (%rdi), %ecx
12. addq $4, %rdi
13. addl %ecx, %eax
14. cmpq %rdx, %rdi
15. jne .L3
16. movl (%rdi), %ecx

CPI == 0.5!
What about “linked list”

Performance determined by the critical path

4 cycles each iteration

5 instructions per iteration

\[ CPI = \frac{4}{5} = 0.8 \]

① .L3:    addq    $8, %rdi
② movq    (%rdi), %rdi
③ addl    $1, %eax
④ testq   %rdi, %rdi
⑤ jne     .L3
What are the characteristics of code with good ILP?

• We’re going to use fiddle to see that!

```c
uint64_t* wide_1(uint64_t threads, uint64_t* data, uint64_t size, uint64_t arg1, uint64_t arg2, uint64_t arg3) {
    register uint64_t i = 0;
    for(i = 0; i < size; i++) {
        i = i+1;
        i = i+1;
        i = i+1;
        i = i+1;
    }
    data[0] = i;
    return data;
}

uint64_t* wide_2(uint64_t threads, uint64_t* data, uint64_t size, uint64_t arg1, uint64_t arg2, uint64_t arg3) {
    register uint64_t a = 4;
    register uint64_t b = 4;
    register uint64_t c = 4;
    register uint64_t d = 4;
    register uint64_t e = 4;
    register uint64_t f = size;
    for(register uint64_t i = 0; i < size; i++) {
        i = i+1;
        a = a+1;
        i = i+1;
        a = a+1;
        i = i+1;
        a = a+1;
        i = i+1;
        a = a+1;
    }
    data[0] = a + b + c + d + e + f;
    return data;
}

uint64_t* wide_5(uint64_t threads, uint64_t* data, uint64_t size, uint64_t arg1, uint64_t arg2, uint64_t arg3) {
    register uint64_t a = 4;
    register uint64_t b = 4;
    register uint64_t c = 4;
    register uint64_t d = 4;
    register uint64_t e = 4;
    register uint64_t f = size;
    for(register uint64_t i = 0; i < size; i++) {
        i = i+1;
        a = a+1;
        b = b+1;
        d = d+1;
        e = e+1;
        i = i+1;
        a = a+1;
        b = b+1;
        d = d+1;
        e = e+1;
        i = i+1;
        a = a+1;
        b = b+1;
        d = d+1;
        e = e+1;
        i = i+1;
        a = a+1;
        b = b+1;
        d = d+1;
        e = e+1;
    }
    data[0] = a + b + c + d + e + f;
    return data;
}
```
Intel SkyLake

Shared L3 $

Core

L2 $
Thread programming
pthreads
The “abstracted” multithreading machine

```c
for(i=0;i<size/4;i++)
    sum += a[i];
for(i=size/4;i<size/2;i++)
    sum += a[i];
for(i=size/2;i<3*size/4;i++)
    sum += a[i];
for(i=3*size/4;i<size;i++)
    sum += a[i];
```
Thread programming

- pthread_t — thread descriptor
- pthread_init() — init a thread descriptor
- pthread_create() — create a thread running a “start_routine” function with “arg” as the argument
  ```
  int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void **(start_routine)(void *),
                   void *restrict arg);
  ```
- pthread_join() — synchronize thread execution
- pthread_mutex_lock, pthread_mutex_unlock — managing a lock
Bounded-Buffer Problem

- Also referred to as “producer-consumer” problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer
We need to control accesses to the buffer!

```c
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = ...;
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
    }
    printf("parent: end\n");
    return 0;
}

void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        // do something w/ item
    }
    return NULL;
}

int buffer[BUFF_SIZE]; // shared global
```
Solving the “Critical Section Problem”

1. Mutual exclusion — at most one process/thread in its critical section
2. Progress — a thread outside of its critical section cannot block another thread from entering its critical section
3. Fairness — a thread cannot be postponed indefinitely from entering its critical section
4. Accommodate nondeterminism — the solution should work regardless the speed of executing threads and the number of processors
Use locks

```c
int buffer[BUF_SIZE]; // shared global
volatile unsigned int lock = 0;

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = ...;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}

void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```
What software thinks about “multiprogramming” hardware

for(i=0; i<size/4; i++)
  sum += a[i];

for(i=size/4; i<size/2; i++)
  sum += a[i];

for(i=size/2; i<3*size/4; i++)
  sum += a[i];

for(i=3*size/4; i<size; i++)
  sum += a[i];

sum = 0

sum = 0

sum = 0

sum = 0

Others do not see the updated value in the cache and keep working — incorrect result!
What happens when we write in coherent caches?

```c
for (i=0; i<size/4; i++)
    sum += a[i];

sum = 0
```

```c
for (i=size/4; i<size/2; i++)
    sum += a[i];

sum = 0
```

```c
for (i=size/2; i<3*size/4; i++)
    sum += a[i];

sum = 0
```

```c
for (i=3*size/4; i<size; i++)
    sum += a[i];

sum = 0
```
Tips of performance thread programming

• Reduce sharing: coherence misses
• Avoid fine-grained locks: serialization of execution
• Avoid short threads: thread spawning overhead
Data level parallelism
SIMD in processors

• SIMD: Single instruction, multiple data
  • Each instruction can perform operations on several datasets concurrently
• Streaming SIMD Extensions (SSE) that allows x86 processor architectures to support “SIMD” execution model
• ARM’s NEON

\[
\begin{pmatrix}
1.0 \\
2.0 \\
3.0 \\
4.0
\end{pmatrix} + \begin{pmatrix}
5.0 \\
6.0 \\
7.0 \\
8.0
\end{pmatrix} = \begin{pmatrix}
6.0 \\
8.0 \\
10.0 \\
12.0
\end{pmatrix}
\]
SIMD hardware

- Vector registers
- Vector load/store unit
- Memory hierarchy

25
Streaming SIMD Extensions (SSE)

- SSE, introduced by Intel in 1999 with the Pentium III, creates eight new 128-bit registers
  - Added 8 128-bit registers — XMM0-XMM7
  - You may use each register to store
    - 2 double-precision floating point numbers
    - 4 single-precision floating point numbers
  - Extended since introduced — SSE2, SSE3, SSE4, SSE4.1, SSE4.2, SSE4a
- They are processor-dependent instructions
  - AMD RyZen supports SSE4a, SSE4.1, SSE4.2
  - intel Core i7 doesn’t support SSE4a
  - VIA Nano only support SSE4.1
Matrix multiplication with SSE4

```c
void vector_blockmm(double **a, double **b, double **c)
{
    int i, j, k, ii, jj, kk, x;
    __m256d va, vb, vc; // compiler would allocate a register as long as these variables can fit
    for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
        for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
            for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
                for(ii = i; ii < i+(ARRAY_SIZE/n); ii++) {
                    for(jj = j; jj < j+(ARRAY_SIZE/n); jj+=VECTOR_WIDTH) {
                        vc = _mm256_load_pd(&c[ii][jj]); // load values into a vector register

                        for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                        {
                            va = _mm256_broadcast_sd(&a[ii][kk]); // load one value & fill the vector register
                            vb = _mm256_load_pd(&b[kk][jj]); // load values into a vector register
                            vc = _mm256_add_pd(vc,_mm256_mul_pd(va,vb)); // vector multiplication
                        }
                        _mm256_store_pd(&c[ii][jj],vc); // store values into a vector register
                    }
                }
            }
        }
    }
}
```
Using OpenMP for both TLP/DLP
Most of time, we want to parallelize just iterations in a loop: OpenMP

- They all start with “#pragma omp”
- “#pragma ...” is a way to add arbitrary extension to C/C++
- #pragma omp parallel for
  - Run the following for loop with multiple threads.
  - The loop needs to be pretty simple.
  - Something like “for(int i = C; i < K; i+=B)”
    - K and B need to fixed for the execution of the loop
    - Otherwise, the compiler will ignore the #pragma
- Apply to one loop
  - Nesting is not productive
  - Use an outer loop – Bigger chunks of work for the threads.
The three #pragmas you'll need for this lab are
- #pragma omp parallel for for parallelizing loops.
- #pragma omp critical for parallelizing loops.
- #pragma omp simd for vectorizing loops
Histogram in OpenMP

```c
extern "C"
uint64_t* run_openmp_histogram(uint64_t thread_count, uint64_t * data, uint64_t size, uint64_t arg1, uint64_t arg2, uint64_t arg3) {
    for(int i =0; i < 256;i++) {
        histogram[i] = 0;
    }

    #pragma omp parallel for
    for(uint64_t i = 0; i < size; i++) {
        for(int k = 0; k < 64; k+=8) {
            uint8_t b = (data[i] >> k)& 0xff;
            #pragma omp critical
            histogram[b]++;
        }
    }
    return data;
}
```
Programming assignment
Matrix exp: $B = A^N$
Continue from Lab 4

• Same problem as last time.
• Now with threads and vectors
• Speedup targets
  • 600 2: 8.5x (vs 2.2 for lab 4)
  • 350 25: 18x (vs 3.5 for lab 4)
  • 120 320: 102.6x (vs 18.5 for lab 4)
  • Overall you need to achieve ~5.5x speedup
• Get your work done in matexp_solution.hpp
• You’re supposed to use OpenMP
  • You may try other available tools on datahub, not supported.