Loop Schedules

Which Thread Executes Which Iterations

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Assigning Iterations to Thread: Default

• The OpenMP runtime is free to assign any iteration to any thread
• A typical default is “static”
  • Assigns a block of contiguous iterations to each thread equally
  • E.g., if there are 800 iterations and 8 threads:
    • Thread 0 executes iteration 0-99, thread 1 executes iteration 100-199, and so on
  • Often, this helps with spatial locality because contiguous iterations tend to access contiguous memory locations
• However, as a programmer you can control this assignment
  • A particular way of assigning iterations to threads is called a “schedule”.
Assigning Iterations to Threads: Example

• How to balance the work per thread when the work per iteration is inherently imbalanced

```c
#pragma omp parallel for private(size)
for(int i=0; i<n; i++) {
    size = f(i);
    if (size < 10)
        smallwork(x[i]);
    else
        bigwork(x[i]);
}
```

Some Iterations are expensive, but we don’t know which because it depends on the function f
Dynamic Schedule

• We can have each thread pick the next iteration available
  • And when it finishes, go back and pick the next available one

• This can be implemented in many ways
  • E.g., having a shared variable that holds the value of the next available iteration

• Idea: every time a thread goes to ask for more work, it is given a chunk of iterations
  • We would like the chunk size to be controlled by the programmer
Schedule Clause and General Form

- schedule (kind[, chunk])
  - kind:
    - static
    - dynamic,
    - guided, (a variant of dynamic)
    - runtime, (programmer can set the schedule at runtime, using `omp_set_schedule(..)`)
    - auto (let the compiler/runtime decide the schedule)
  - Optional chunk = scalar integer value
Schedule: Static, Dynamic or Guided

• **Static**: iterations are divided as evenly as possible among all threads with each thread getting a contiguous range of iterations

• **Static, chunk**: iterations are divided into chunks of size \( \text{chunk} \). Chunks are then assigned in round robin fashion to threads – *interleaved*

• **Dynamic, chunk**: iterations are divided into chunks of size \( \text{chunk} \) (1 if unspecified) and are assigned to threads dynamically after an initial round robin assignment – *simple dynamic*

• **Guided, chunk**: chunk size decreases exponentially from an implementation dependent value to \( \text{chunk} \) (1 if unspecified). Chunks are assigned dynamically – *guided self scheduling*
Tradeoffs in Static vs. Dynamic Scheduling

• Static has lower synchronization overhead, better spatial locality, but cannot deal with imbalances well

• Dynamic schedule will balance load well, but
  • Has a higher synchronization overhead, as threads have to coordinate frequently to decide assign iterations to threads
    • I.e., there will be overhead in picking the next available iteration
  • Spatial locality:
    • As iterations are assigned to threads when they ask for work, consecutive (chunk of) iteration may go to different threads, reducing or destroying spatial locality
Assigning Iterations to Threads

• Use dynamic schedule to balance the work per thread when the work per iteration is inherently unbalanced

```c
#pragma omp parallel for schedule(dynamic,16)
for(int i=0; i<n; i++) {
    int size = f[i];
    if (size < 10)
        z[i] = smallwork(x[i]);
    else
        z[i] = bigwork(x[i]);
}
```

L.V.Kale
For loop inside `omp parallel`

Work sharing within `parallel`

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Worksharing with `omp for`

- We will introduce a directive
  - `#pragma omp for`
  - It is one of the “worksharing” constructs

- This can be used only inside a ”omp parallel” block
  You already have a team of threads, which is being told to share the work of a loop

- The following two are equivalent

```
#pragma omp parallel
#pragma omp for
```
```
#pragma omp parallel for
```
Recall: Simple Example with parallel

```c
for (i=0; i<N; i++) {
    A[i] = x*B[i];
}
```

```c
#pragma omp parallel
{ 
    int id = omp_get_thread_num();
    int p = omp_get_num_threads();
    int mystart = (N*id)/p;
    int myend = (N*(id+1))/p;
    if(id == (p-1)) myend = N;
    for( i=mystart; i<myend; i++ ) {
        A[i] = x*B[i];
    }
}
```

```c
#pragma omp parallel for
for( i=0:N; i<N; i++) {
    A[i] = x*B[i];
}
```

```c
#pragma omp parallel
#pragma omp for
for( i=0:N; i<N; i++) {
    A[i] = x*B[i];
}
```

This a more flexible construct ... you can have multiple blocks of code, including some “workshared” loops, others normal SPMD
```c
#pragma omp parallel
{
  ..
  SPMD code block A executed by all threads
  ..
  #pragma omp for
  forLoop1
  #pragma omp for schedule(dynamic)
  forLoop2
  ..
  SPMD code block B executed by all threads
  ..
}
```
Beyond Loops: The parallel Construct

SPMD Parallelism

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
parallel for: Restrictions

• The number of times that the loop body is executed (trip-count) must be available at runtime before the loop is executed

• The loop body must be such that all iterations of the loop are completed

• Also, some computations may be parallel, but not expressible easily via parallel loops
The **parallel** Directive

- This allows the programmer to explicitly say what each thread does

  ```
  #pragma omp parallel [clause [clause ...]]
  structured block
  ```

- This directive tells all the threads to execute the structure block
- What is the point of having all threads execute the same code?
- Each thread has access to each id and so it can do different work depending on id
  - Ids are serial numbers from 0 to total number of thread
The **parallel** Directive:

• When a *parallel* directive is encountered, threads are spawned, which execute the code of the enclosed structured block (the parallel region)

• The number of threads can be specified just like for the *parallel for* directive

• Each thread executes the code in the enclosed code-block

• You can specify private variables as before (but no “lastprivate”)
**parallel vs. parallel for**

- Arbitrary structured blocks v/s loops
- Coarse grained v/s fine grained
- Replication v/s work division

```c
#pragma omp parallel for
for(i=0; i<10; i++) {
    printf("Hello world\n");
}
```

Output: 10 Hello world messages

```c
#pragma omp parallel private(i)
{
    for(i=0; i<10; i++) {
        printf("Hello world\n");
    }
}
```

Output: 10*T Hello world messages where T = number of threads
A Simple Example with `parallel`

```c
for( i=0; i<N; i++ ) {
    A[i] = x*B[i];
}
```

```c
#pragma omp parallel
{ int id = omp_get_thread_num();
  int p = omp_get_num_threads();
  int mystart = (N*id)/p;
  int myend = (N*(id+1))/p;
  if(myend>N) myend = N;
  for( i=mystart; i<myend; i++ ) {
    A[i] = x*B[i];
  }
}
```

Each thread calculates the range of iterations it’s going to work on using variables `mystart` and `myend` after finding out
- The total number of threads, `p`, in its team and
- Its own rank, `id`, within the team
**parallel vs. parallel for**

- The code with parallel is much longer and complicated
- But it gives you, the programmer, much greater control
  - And the cases where you will use it are typically much longer pieces of code
- To use the full power of this construct you will also need to use other constructs, particularly synchronization constructs
parallel Construct: Example 2

double x, y;
int i, j, m, n, maxiter, depth[300][200],
    dith_depth[300][200], mandel_val();
n = 300; m = 200;
maxiter = 200;

#pragma omp parallel for private(j, x, y)
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
        x = i/ (double) m;
        y = j/ (double) n;
        depth[j][i] = mandel_val(x, y, maxiter);
    }

#pragma omp parallel for private(j)
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++)
        dith_depth[j][i] = 0.5*depth[j][i]
                    + 0.25*(depth[j-1][i] + depth[j+1][i])

Parallelizing consecutive for loops requires initializing threads twice!
parallel Construct: Example 2 Rewritten

• Rewrite with parallel

```c
double x, y;
int i, j, m, n, maxiter, depth[300][200],
    dith_depth[300][200], mandel_val();
n = 300; m = 200;
maxiter = 200;
int id, myfirst, mylast;
omp_set_num_threads(16);
#pragma omp parallel private(id, myfirst, mylast, i, j, x, y)
{
    id = omp_get_thread_num();
    myfirst = first_idx(id, m, 16);
    mylast = last_idx(id, m, 16);
    for (i = myfirst; i <= mylast; i++)
        for (j = 1; j <= n; j++) {
            x = i/ (double) m;
            y = j/ (double) n;
            depth[j][i] = mandel_val(x, y, maxiter);
        }
    for (i = myfirst; i <= mylast; i++)
        for (j = 1; j <= n; j++)
            dith_depth[j][i] = 0.5*depth[j][i]
                              + 0.25*(depth[j-1][i] + depth[j+1][i]);
}
```

Threads are forked once and then work independently
Synchronization Constructs

Critical, Atomic, and Locks
Synchronization – Motivation

• Different threads need to coordinate with each other in a parallel program

• They communicate by writing and reading to the same shared variables
  • But they may step on each other’s toes
  • E.g., simultaneously trying to write to the same variable

• Also, sometimes we need to signal from one thread to the other
  • Signaling, for example, that some value is ready for the other thread to use
Types of Synchronization

• Concurrent access to shared data may result in data inconsistency — mechanism required to maintain data consistency: mutual exclusion

• Sometimes code sections executed by different threads need to be sequenced in some particular order: event synchronization
  • From one thread to the others
  • Wait for all threads to complete some section of code before continuing on
    • This is called a barrier
Mutual Exclusion

- Mechanisms for ensuring the consistency of data that is accessed concurrently by several threads
  - Critical directive
  - Atomic directive
  - Library lock routines
Critical Section: Syntax

#pragma omp critical [name]
    structured_block

- No other thread is allowed to execute any code inside any critical section in the program if one thread is inside it
- Other threads, if they encounter this directive, are made to wait, and only one of them will proceed into the critical section once the thread inside leaves
- The critical sections may be in many places in the code
- If you add a name, then the restriction applies only to those sections that share the name
Global vs. Named Critical Sections

• Access to unnamed critical sections is synchronized with accesses to all critical sections in the program: global lock

• To change the global lock behavior use the optional name parameter – access to a named critical section is synchronized only with other accesses to critical sections with the same name
Only one thread can be inside any of S1, S2, or S3

At most one thread can be inside any of S2 or S3, and separately, another thread can be inside S1
cur_max = MINUS_INFINITY;
#pragma omp parallel for
for(i=0; i<n; i++){
    ...
    if (a[i] > cur_max)
        cur_max = a[i];
    ...
}
Critical Section Example

cur_max = MINUS_INFINITY;
#pragma omp parallel for
for(i=0; i<n; i++){
    ...
    #pragma omp critical
    {
        if (a[i] > cur_max)
            cur_max = a[i];
    }
    ...
}
cur_max = MINUS_INFINITY; maxIndex = -1;
#pragma omp parallel for
for(i=0; i<n; i++){
    ...
#pragma omp critical
    {
        if (a[i] > cur_max)
            { cur_max = a[i]; maxIndex = i}
    }
    ...
}
Synchronization Constructs

Critical, Atomic, and Locks

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Atomic Directive

• Critical sections are expensive (we will see later why)
  • But processors support fast single-variable updates as if they are in critical
  • Atomic directive provides access to those hardware capabilities when available
• The body of an atomic directive is a single assignment statement
• There are restrictions on x, expr, operator, and binop – these restrictions ensure that the assignment statement can be translated into an atomic sequence of machine instructions to read, modify, and write the memory location for x
• Syntax:

```
#pragma omp atomic
<Restricted Statement form>
```

Restricted statement form can be one of:
- x++; ++x; x--; --x;
- x binop = expr

binop: binary operator, such as +, -
Variants of Atomic Directive

• Many other variants of atomic are supported
• They typically correspond to what the hardware can support as atomic operations
  • I.e., an atomic operation completes before another atomic operation can start on the same variable (no interleaving)
• We will cover only one variant defined by the capture clause
Atomic Directive with the capture Clause

• We want to atomically change a variable and “capture” its old (or new) value into another variable

• Syntax: two forms are allowed

```
#pragma omp atomic capture
<Restricted expression statement>
```

```
#pragma omp atomic capture
<Restricted Structured Block>
```

<Restricted expression statement> can be one of:

```
v = x++; 
v = ++x;
v = x--; 
v = --x; 
v = x binop = expr; etc...
```

Restricted Structured block can be:

```
{ v = x; x binop = expr}
{x binop = expr; v =x}
```

For additional forms see OpenMP 4.5 Standard
For a particular compiler see:
https://software.intel.com/en-us/node/524509
Atomic and Critical

These two formulations are equivalent
But the one with atomic is more efficient
Library Lock routines

• Routines to:
  • In the following, `plock` is a pointer to a variable of type `omp_lock_t`
  • Create a lock
    • `omp_init_lock(plock)`
  • Acquire a lock, waiting until it becomes available if necessary
    • `omp_set_lock(plock)`
  • Release a lock, resuming a waiting thread, if one exists
    • `omp_unset_lock(plock)`
  • Try and acquire a lock but return instead of waiting if not available
    • `omp_test_lock(plock)` (returns true if lock acquired)
  • Destroy a lock
    • `omp_destroy_lock(plock)`
Library Lock Routines

• Locks are the most flexible of the mutual exclusion primitives because there are no restrictions on where they can be placed

• The previous routines don’t support nested acquires – deadlock if tried!! – a separate set of routines exist to allow nesting
  • I.e. With nesting, the same thread may acquire the lock multiple times,
    • Each time a count is incremented, and unlock decrements the count
    • Only when the count reaches back to 0, can other threads lock the same variable
Mutual Exclusion Features

• Apply to critical, atomic as well as library routines:
  • NO fairness guarantee
  • Guarantee of progress
  • Careful when using multiple locks – lots of chances for deadlock
• Deadlocks: when a bunch of threads are waiting for each other in a circular fashion
  • Waits arise from locks held by others
  • Thread 1 locks A, requests B, while Thread 2 locks B, requests A
  • More generally: a circular wait: thread 1 waiting for thread 2, which waits for thread 3, which waits for thread 1
Case Study: Finding Primes

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Example: List of Primes

• Consider the problem of finding the first million prime numbers
  • We want to output an array containing the first million primes

• Note that \texttt{parallel for} requires that the loop bounds need to be known before the loop starts
• So, how can we parallelize this?

Sequential Code:

\begin{verbatim}
x = 3; j = 1; primes[0] = 2;
while(j < 1000000){
    if(isPrime(x)){
        primes[j] = x;
        j++;
    }
    x = x + 2;
}
\end{verbatim}
Ideas for Parallelizing PrimesList

• We can have each thread explore the next unexplored odd integer beginning with 3
• Both x and j need to be protected because multiple threads want to read and write them
• We can use locks or atomic variables for this purpose

```
x = 3; j = 1; primes[0] = 2;
while(j < 1000000){
    if(isPrime(x)){
        primes[j] = x;
        j++;
    }
    x = x + 2;
}
```
Version 1: Parallelizing PrimesList

• We protect writes to j and x using atomic

• Note that between the time a thread starts testing for x and the time it increments x, x may have been changed by other threads
  • So, after finishing testing 3, a thread may start working on testing 21, if other threads have already taken the number in between

```c
int x,j;
x = 3; j = 1; primes[0] = 2;
#pragma omp parallel
while(j < 1000000){
    if(isPrime(x)){
        primes[j] = x;
        #pragma omp atomic
        j++;
    }
    #pragma omp atomic
    x = x + 2;
}
```

Does this work?
Version 1: Parallelizing PrimesList

• The problem is between the time we test the primality of a number (x), and the time that it executes the assignment statement, some other thread might have changed the value of x

• Also, two different threads may try to assign with the same value of j

```
int x, j;
x = 3; j = 1; primes[0] = 2;
#pragma omp parallel
while (j < 1000000) {
    if (isPrime(x)) {
        primes[j] = x;
        #pragma omp atomic
        j++;
    }
    #pragma omp atomic
    x = x + 2;
}
```

Does this work?
Version 2: Parallelizing PrimesList

• A thread saves the current value of x and increases it by 2 in a single critical section
  • No other thread can interfere
• A thread atomically increments j but saves the new value in its private variable k

```c
int x,j,myX,k;
x = 3; j = 0; primes[0] = 2;
#pragma omp parallel private(myX,k)
while(j < 1000000){
#pragma omp atomic capture
{ myX = x; x = x + 2;}
if(isPrime(myX)){
#pragma omp atomic capture
  k = j++;
  primes[k] = myX;
}
}
```

Does this work?
Version 2: Parallelizing PrimesList

• This almost works
• What are the problems?
• Is the primes array sorted?
  • No, although it is almost sorted
  • Some threads might run ahead
• How do we stop after the first one million primes?
  • While one thread is working on testing the millionth prime, another thread might finish testing the next prime and add it to the list

Fixing this is left as an exercise for you

```c
int x,j,myX,k;
x = 3; j = 0; primes[0] = 2;
#pragma omp parallel private(myX,k)
while(j < 1000000){
    #pragma omp atomic capture
    { myX = x; x = x + 2;}
    if(isPrime(myX)){
        #pragma omp atomic capture
        k = j++;
        primes[k] = myX;
    }
}
```
Parallelizing PrimesList: Ideas for Fixes

• Let the loop go further for a few more iterations
  • How many?
    • Maybe $j < 1000000 + \text{numThreads}$?

• Sort the array at the end?
  • Too expensive
  • And it’s mostly sorted

```c
int x, j, myX, k;
x = 3; j = 0; primes[0] = 2; #pragma omp parallel private(myX, k)
while(j < 1000000){
  #pragma omp atomic capture
  { myX = x; x = x + 2; }
  if(isPrime(myX)){
    #pragma omp atomic capture
    k = j++;
    primes[k] = myX;
  }
}
```
Additional Coordination Constructs

Barrier, Single, and Master Directives
Parallel Sections

• Provide another way of creating a team of threads
  • In addition to construct parallel or parallel for
• From the Openmp 4.5 standard:

```
#pragma omp sections {clause [,] clause} ... } new-line
{
  [#pragma omp section new-line]
  structured-block
  [#pragma omp section new-line
  structured-block]
  ...
}
```

• Independent different pieces of code assigned to different threads
• (BTW: openmp standard is the (semi-)final arbiter
  • http://www.openmp.org/specifications/
  • Final arbiter is of course your compiler ... hopefully it implements the latest standard ... check always
barrier Construct: Making Everyone Wait

• This can be thought of as an event synchronization construct

```c
#pragma omp barrier
```

• No thread can pass the barrier directive unless all threads (in the current team) have arrived at it

• The programmer must take care to ensure all threads (in the team) encounter this statement or none of them do, for every execution of the program
The master Construct

• In a parallel region, sometimes you want some action to be done only by the master thread
  • The parallel region may be a “parallel for” or a “parallel” construct, for example

• Syntax:  

```plaintext
#pragma omp master
structured_block
```

• The master thread executes the structured_block, while

• All the other threads pass past it
  • I.e., they do not execute the structured_block nor do they wait for the master thread to execute it
The **single** Construct

- Similar in spirit to the master construct
- In a parallel region, sometimes you want some action to be done only by a single thread
  - It doesn't matter which thread executes it
- Syntax: `#pragma omp single structured_block`
- The first thread to arrive at this directive executes the `structured_block`, while
- All the other threads pass past it
  - I.e., they do not execute the `structured_block` nor do they wait for execution of this `structured_block` by the first thread
Example: Prefix Sum

Recursive Doubling with Barriers
Prefix Sum Problem

• Given array $A[0..N-1]$, produce $B[N]$, such that $B[k]$ is the sum of all elements of $A$ up to $A[k]$

$$
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
3  & 1  & 4  & 2  & 5  & 2  & 7  & 4 \\
\end{array}
$$

Prefix Sum: A good Sequential Algorithm

• Data dependency from iteration to iteration
  • How can this be parallelized at all?

```plaintext
B[0] = A[0];
for (i=1; i<N; i++)
  B[i] = B[i-1] + A[i];
```

• It looks like the problem is inherently sequential, but theoreticians came up with a beautiful algorithm called recursive doubling or just parallel prefix
Parallel Prefix: Recursive Doubling

- N Data Items
- P Processors
- N = P

Log P Phases
- P additions in each phase
- P log P operations
- Completes in O(logP) time
OpenMP Formulation for Parallel Prefix

• We don’t have \( n \) processors
  • I.e., the number of threads will be much smaller than the size of the data array

• So, we will simulate \( n \) processors using \( p \) threads

• Notice that each phase of the computation must finish before the next phase of the computation starts
  • We will use OpenMP’s `barrier` directive for that

• Basic description of actions in each phase with distance \( d \)
  • Every “processor” \( i \) adds its value to the value held by a processor distance \( d \) away
  • Simulation: \( B[i+d] += B[i] \), but you have to be careful to avoid dependencies
    • I.e., copy \( B[i] \) into a temporary variable at \( i+d \), say \( C[i+d] \) and then add \( C[i] \) to \( B[i] \) for every \( i \)
  • Note \( d \) doubles in every phase
Parallel Prefix: Recursive Doubling

L.V. Kale
#pragma omp parallel for
for(i=0;i<n;i++){B[i]=A[i];}
int d=1;
while(d<n)  // this loop will run for lg n steps
{
    int i;
    #pragma omp parallel for
    for(i=d;i<n;i++)C[i]=B[i-d];

    #pragma omp parallel for
    for(i=d;i<n;i++)B[i]+=C[i];

    d*=2;
}

...
Critique of Prefix Algorithm 1

• The sequential algorithm had \( n \) additions
• But the parallel algorithm is doing a total of \( n \times (\log n) \) additions
  • Although they are parallelized by \( p \) threads
  • This is an example of an algorithm that is not “work efficient”
• It uses \( \log n \) barriers, which are expensive operations
• Maybe a thread oriented approach will avoid the \( \log n \) factors
Prefix Sum Algorithm 2: A Thread Oriented Approach

• What if we let each thread calculate prefix sum over its own range of array?
  • I.e., thread id is responsible for range \( B[\frac{n \times \text{id}}{p} : \frac{n \times (\text{id}+1)}{p} - 1] \)
  • Id : my thread’s serial number; \( p \) : total number of threads
  • Assuming \( n \) is a multiple of \( p \)

• But then each thread needs the sum of all numbers to its left
Thread\textsubscript{0} \quad \cdots \quad \text{Thread}_{\text{id}-1} \quad \text{myBegin} \quad \text{Thread}_{\text{id}} \quad \text{myEnd} \quad \cdots
Prefix Sum Algorithm 2: A Thread Oriented Approach

• What if we let each thread calculate prefix sum over its own range of array?
  • I.e., thread i is responsible for range $B_{\frac{n \times i d}{p} : \frac{n \times (i d+1)}{p} - 1}$
    *id – my thread’s serial number; *p – total number of threads
    Assuming n is a multiple of p
• But then each thread needs the sum of all numbers to its left
• If we are willing to double the amount of work, we can obtain this sum with a much smaller prefix sum problem of size p
  1. First loop: every thread calculates sum s over its sub-range and copies s into a shared array called sums at $\text{sums}[i d]$
  2. Calculate prefix sum of the sums array
    • $\text{sums}[i d-1]$ has the sum of all values to the left of thread numbered id
  3. Second loop: every thread with serial number id calculates the prefix sum in array B using $\text{sums}[i d-1]$ and the values in A
Thread_0  Thread_{id-1}  Thread_{id}

sums

sums_{id-1}

\text{myBegin}

\text{myEnd}

\text{Id-1}
... omp_set_num_threads(p);

#pragma omp parallel
{
    int id=omp_get_thread_num();
    int myBegiN = (n*id)/p;
    int myEnD = min( (n*id+1)/p, n);

    int sum=0;
    for(int i=myBegiN;i<myEnD;i++)
        sum+=B[i];
    sums[id]=sum;

    #pragma omp barrier
    #pragma omp single
    {
        for(int i=1;i<p;i++)
            sums[i]+=sums[i-1];
    }

    if(id>0)B[myBegiN]+= sums[id-1]
    for(int i=myBegiN+1;i<myEnD/p;i++)
        B[i]+=B[i-1];
}

int sum=0;
for(int i=myBegiN;i<myEnD;i++)
    sum+=B[i];
sums[id]=sum;
#pragma omp barrier
#pragma omp single
{
    for(int i=1;i<p;i++)
        sums[i]+=sums[i-1];
}
if(id>0)B[myBegiN]+= sums[id-1]
for(int i=myBegiN+1;i<myEnD/p;i++)
    B[i]+=B[i-1];

Form Local sum
• Calculate Prefix sum of size p
• Sums [id] now contains the sum of values of all previous threads’ ranges
• This can be done in parallel but it’s not worth it

Complete the Prefix sum
Example: Prefix Sum

Recursive Doubling with Barriers: Algorithm 2
Prefix Sum: A good Sequential Algorithm

• Data dependency from iteration to iteration
  • How can this be parallelized at all?

\[
\begin{align*}
  B[0] &= A[0]; \\
  \text{for (} i=1; i<N; i++ \text{)} & \quad B[i] = B[i-1] + A[i];
\end{align*}
\]

• It looks like the problem is inherently sequential, but theoreticians came up with a beautiful algorithm called recursive doubling or just parallel prefix
Parallel Prefix: Recursive Doubling

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Log P Phases
- P additions in each phase
- P log P operations
- Completes in $O(\log P)$ time

N Data Items
P Processors
N=P
Initialize B with values from A

C[k] temporarily stores the value that we want to add to B[k]
Critique of Prefix Algorithm 1

• The sequential algorithm had \( n \) additions
• But the parallel algorithm is doing a total of \( n \times (\log n) \) additions
  • Although they are parallelized by \( p \) threads
  • This is an example of an algorithm that is not “work efficient”
• It uses \( \log n \) barriers, which are expensive operations
• Maybe a thread oriented approach will avoid the \( \log n \) factors
Prefix Sum Algorithm 2: A Thread Oriented Approach

• What if we let each thread calculate prefix sum over its own range of array?
  • I.e., thread id is responsible for range $B[\frac{n\times id}{p} : \frac{n\times (id+1)}{p} - 1]$
  • Id : my thread’s serial number; p : total number of threads
  • Assuming n is a multiple of p
• But then each thread needs the sum of all numbers to its left
Prefix Sum Algorithm 2: A Thread Oriented Approach

• What if we let each thread calculate prefix sum over its own range of array?
  • I.e., thread $i$ is responsible for range $B_{\frac{n \times id}{p} : \frac{n \times (id+1)}{p} - 1}$
    *id – my thread’s serial number; *p – total number of threads
  Assuming $n$ is a multiple of $p$

• But then each thread needs the sum of all numbers to its left

• If we are willing to double the amount of work, we can obtain this sum with a much smaller prefix sum problem of size $p$
  1. First loop: every thread calculates sum $s$ over its sub-range and copies $s$ into a shared array called sums at $\text{sums}[id]$
  2. Calculate prefix sum of the sums array
    • $\text{sums}[id-1]$ has the sum of all values to the left of thread numbered $id$
  3. Second loop: every thread with serial number $id$ calculates the prefix sum in array $B$ using $\text{sums}[id-1]$ and the values in $A$
omp_set_num_threads(p);

#pragma omp parallel
{
    int id=omp_get_thread_num();
    int myBegin = (n*id)/p;
    int myEnd = min( (n*(id+1))/p, n);

    int sum=0;
    for(int i=myBegin;i<myEnd;i++)
        sum+=B[i];
    sums[id]=sum;

    #pragma omp barrier
    #pragma omp single
    {
        for(int i=1;i<p;i++)
            sums[i]+=sums[i-1];
    }
    #pragma omp barrier

    if(id>0)B[myBegin]+= sums[id-1]
    for(int i=myBegin+1;i<myEnd; i++)
        B[i]+=B[i-1];
}

Form Local sum
• Calculate Prefix sum of size p
• Sums [id] now contains the sum of values of all previous threads’ ranges
• This can be done in parallel but it’s not worth it

Complete the Prefix sum
More Synchronization

Sequential Consistency, and the Flush Directive

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Memory Consistency Issues and OpenMP

• Let’s start with an old “schematic” of a computer
• Instruction memory, CPU, and data memory
• Each arithmetic instruction is like: “add m1, m2, m3”
  • m1,m2,m3 are memory locations
  • Bring contents of m1 and m2 to ALU, add them up, and store the result in m3
• In addition, there are branch instructions
If Processors Remained That Simple ...

- If parallel computers with shared memory were built from such processors
  - Parallelism would be simpler
  - But they’d be slow
- Computer architects added caches, as we know
  - Some variables may exist in caches as well as memory
  - Cache coherence protocols, such as MESI for snoopy caches, handle data in caches
- But also, they added registers
Complexity of Real Computers: Registers

• Real computers became more complex before they became parallel
  • Remember: mostly to deal with slow memory
• Around 1985, the RISC revolution led to load-store architectures
  • Instructions either:
    • Do arithmetic/logical operations on registers, storing the result in registers,
    • Do a load from memory into register, OR
    • Do a store from register into memory
Complexity of Real Computers: Registers

• Real computers became more complex before they became parallel
  • Remember: mostly to deal with slow memory

• Around 1985, the RISC revolution led to load-store architectures
  • Instructions either:
    • Do arithmetic/logical operations on registers, storing the result in registers,
    • Do a load from memory into register, OR
    • Do a store from register into memory

• Variables can be stored in registers
  • But the association between variables and registers is loose
    • Not visible to the hardware
  • May be “spilled” lazily to memory via “write buffers”
Write Buffers

• For sequential processors, how long should a “Store” instruction take?
• You can speed it up by using a write buffer between the CPU and cache
• But this becomes problematic in parallel
Complexity: Compilers Can Reorder Statements

• Remember, compilers were written for sequential processors

• For example it may transform the first block of code below to the second block, by reordering statements
  • Why? Many reasons.. E.g. reducing the number of registers used, or eliminating a pipeline bubble

```c
x = e1; // e1 doesn’t contain y
y = e2; // e2 doesn’t contain x
```

```c
y = e2; // e2 doesn’t contain x
x = e1; // e1 doesn’t contain y
```
Complications with Parallel Processors

• Caches were handled using the extra hardware (snoopy cache controllers)
• But:
  • Data in registers
  • Data in “write buffers” (on its way to memory)
  • Instructions reordered by compiler
• When used with parallel processors, these tend to destroy our intuitive understanding of how parallel processors should behave, especially wrt memory
  • Notions of causality, program order, happens-before relation
• Our intuition is captured by the formalized notion of “sequential consistency”
  • Corresponds to the the simple picture of processor and memory we sketched earlier
Sequential Consistency

• This is a “desired property” of parallel programming systems
• The effect of executing a program consisting of k threads should be the same as some arbitrary interleaving of statements executed by each thread, executed sequentially

Modern processors do not satisfy sequential consistency!
Arbitrator
Initially: $x$, Flag, are both 0

Thread 0:

$x = 25;\\$

Flag = 1;

Thread 1:

while (Flag == 0) ;

Print $x$;

What should get printed?
How to deal with lack of sequential consistency?

• Solution? Give up on registers and write buffers? No way!
• Various complicated processors specific synchronization primitives
• OpenMP provides simple machine-independent \textit{flush} primitives
#pragma omp parallel {
    if (omp_get_thread_num() == 0) {
        //Thread 0: Producer
        Data = computeData();
        flag = 1;
    }
    if (omp_get_thread_num() == 1) {
        //Thread 1: Consumer
        while (flag == 0) {
        }
        print Data;
    }
}
#pragma omp parallel {
    if (omp_get_thread_num() == 0) {
        //Thread 0: Producer
        Data = computeData();
        #pragma omp flush (Data)
        flag = 1;
        #pragma omp flush (flag)
    }
    if (omp_get_thread_num() == 1) {
        //Thread 1: Consumer
        while (flag == 0) {
            #pragma omp flush (flag)
        }
        print Data;
    }
}

L.V.Kale
Naming Variables in the Flush Directive

• If no variables are named, all are flushed
  • I.e., the processor waits until all writes from it in the past are visible and all registers are stored out to memory/cache (spilled, flushed)

• `flush` is especially useful for point-to-point synchronization
  • i.e. for one thread to signal to another (that some event has happened or some data is ready)
More Synchronization II

ordered Directive

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Some Loops Are “Almost” Parallel

B[0] = 1.0;
for (i=1; i< N; i++) {
    A[i] = f(C[i]); //assume f is expensive
    B[i] = B[i-1] +i*A[i];
    D[i] = B[i]*B[i];
}
### The *ordered* Clause

- Original concept of “DoAcross” loops
- Can be used inside a parallel loop

```c
#pragma omp ordered

Code-block
```

- Makes the block of code wait for previous iteration to finish its *ordered* block
- How will you fix the code from the previous page?

```c
B[0] = 1.0;
for (i=1; i< N; i++) {
    A[i] = f(C[i]); // assume f is expensive
    B[i] = B[i-1] + i*A[i];
    D[i] = B[i]*B[i];
}
```
Using **ordered** Directive and **ordered** clause

```c
B[0] = 1.0;  
#pragma omp parallel for ordered
for (i=1; i< N; i++) {
    A[i] = f(C[i]); //assume f is expensive
    #pragma omp ordered
    { B[i] = B[i-1] +i*A[i]; }
    D[i] = B[i]*B[i];
}
```

Computation of f and A[i] happen in parallel
- Note that all iterations use old value of B[i] for computing B
- Old: from before the for loop
- So, it’s ok to do those in parallel

Computation of D[i]s are parallel
Especially useful if it were expensive:
  e.g., D[i] = g(B[i]);

Computation of B[i]s are serialized

L.V.Kale
**depends clause with ordered directive**

- Allows you to specify a dependence in a more general and precise way
  - E.g. Execute the following statement in the current iteration \(i\) after another specific statement in iteration \(i-3\)

```c
#pragma omp parallel for ordered(1)
for (i=1; i< N; i++) {
    codeBlock_A
    #pragma omp ordered depend(sink:i-3)
    {codeBlock_B}
    codeBlock_C
    #pragma omp ordered depend(source)
    codeBlock_D
}
```
depends clause with ordered directive: 2

• Dependences across deeper loop nests can be specified too.

```
#pragma omp parallel
#pragma omp for ordered(2)
for (i=1; i<N; i++) {
    for (j=1; j<M; j++) {
        A[i][j] = foo(i, j);
        #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
        B[i][j] = bar(A[i][j], B[i-1][j], B[i][j-1]);
        #pragma omp ordered depend(source)
        C[i][j] = baz(B[i][j]);
    }
}
```

Adapted From OpenMP Application Programming Interface Examples
At https://www.openmp.org/specifications/
Copyright © 1997-2016 OpenMP Architecture Review Board.