

# Shared Memory Programs

## ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

## Sorting and array with Pthread

- Consider an array of  $n$  elements to sort
- Let's say you have a machine with 2 cores
- One approach is to split the array in two among two threads
  - Each sorting can be done in  $O(n \log n)$
  - Then merging is in  $O(n)$
  - Therefore, if the array is large, one should get close to a speedup of 2 because the sorting (which is done in parallel) is the dominant operation
    - But we know by Amdahl law that for medium-sized arrays we could really be hurt by the sequential merge
  - Note that we do not need for any mutual exclusion here, because we know that we're sorting disjoint pieces of the array

## Sorting: Pthread

6 3 2 9 1 4 8 7 5 0

each worker thread looks at its half of the array

6 3 2 9 1 4 8 7 5 0

each worker thread **sorts** its half **in place**

1 2 3 6 9 0 4 5 7 8

the master thread **merges** the array (perhaps in place)

0 1 2 3 4 5 6 7 8 9

## Sorting in pthreads

- Let's assume that  $n$  is even
- Let's assume we have a `merge()` function

```
void merge(int *array, int size);
```
- The main routine looks like:

```
int main(int argc, char **argv) {
    int A[N]; // Array of size N
    // create the two worker threads to do the work
    // wait for the two worker threads
    merge(A, N);
}
```
- I will not check return values, but you always should!

## Sorting in pthreads

- Code for the worker threads

```
struct {
    int *array;
    int size;
} argstruct;

void *do_work(void *arguments) {
    struct argstruct *args = (struct argstruct *)arguments;

    qsort(args->array, args->size, sizeof(int), intcompare);
    return NULL;
}

// helper function
int intcompare(const void *x, const void *y) {
    if (*(int*)x > *(int*)y) return 1;
    if (*(int*)x < *(int*)y) return -1;
    return 0;
}
```

## Sorting in pthreads

- Code for the master thread

```
struct {
    int *array;
    int size;
} argstruct;

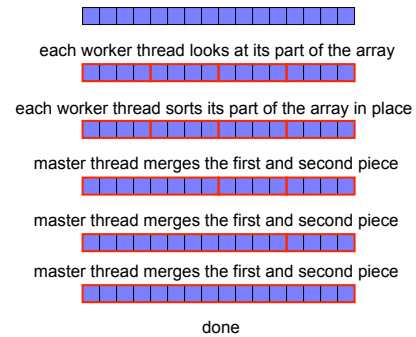
int main() {
    int A[N]; // Array of size N
    struct argstruct *arg1, *arg2;
    pthread_t thread1, thread2;
    arg1 = (struct argstruct *)calloc(1, sizeof(struct argstruct));
    arg2 = (struct argstruct *)calloc(1, sizeof(struct argstruct));
    arg1->size = N/2; arg1->array = A;
    arg2->size = N/2; arg2->array = &A[N/2];
    pthread_create(&thread1, NULL, do_work, arg1);
    pthread_create(&thread2, NULL, do_work, arg2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    merge(A, N);
    free(arg1); free(arg2);
}
```

## What about using more threads?

- What about using more threads to exploit more processors/cores?
- One possibility: cut the array in T pieces, where T is the number of threads
- Drawbacks:
  - The merge function is sort of messy to write
  - And it has higher complexity

## What about using more threads?



## What about using more threads?

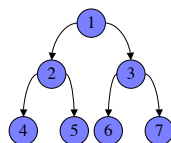
- Let n be the size of the array, and p the number of processors
- Assume p divides n
- The (worst-case) complexity of the merging is approximately  $2^n/p + 3^n/p + \dots + p^n/p = O(n^*p)$
- If p is large and n is huge, this could be bad
- Amdahl's law tells us that even a small sequential part can be bad
  - In this case it may not even be that small
  - Let's parallelize it

## Multi-threaded Merging?

- One solution is to write a multi-threaded merge routine that does merges in parallel
  - would take as input A, n, and p.
  - would use p threads
- This is not very elegant because
  - One creates p threads to do the sorting
  - We wait until everything is sorted
  - We terminate the p threads
  - We create p new threads to do the merging
- A more elegant implementation is to do the partial sorting and partial merging all at the same time recursively

## Recursive multi-threading

- Create a function that does the sorting of one array by
  - creating two threads to do partial sorting
  - doing the merging
- The threads doing the partial sorting call this function, and thus can create threads themselves



a binary tree of threads

## Recursive multi-threading

```

struct {
    int *array;
    int size;
} argstruct;

void *do_work(void *arguments) { // recursive do_work
    pthread_t thread1, thread2;
    struct argstruct *args = (struct argstruct *)arguments;
    struct argstruct *arg1, *arg2;
    pthread_t thread1, thread2;

    if (args->size < 20) { // small array
        qsort(args->array, args->size, sizeof(int), intcompare);
        return NULL;
    }

    arg1 = (struct argstruct *)calloc(1, sizeof(struct argstruct));
    arg2 = (struct argstruct *)calloc(1, sizeof(struct argstruct));
    arg1->size = args->size / 2; arg1->array = args->array;
    arg2->size = args->size / 2; arg2->array = &(A[args->array/2]);

    pthread_create(&thread1, NULL, do_work, arg1);
    pthread_create(&thread2, NULL, do_work, arg2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    merge(args->array, args->size);
    return NULL;
}
    
```

## Small Optimization

```
struct {
    int *array;
    int size;
} argstruct;

void *do_work(void *arguments) { // recursive do_work
    pthread_t thread1, thread2;
    struct argstruct *args = (struct argstruct *)arguments;
    struct argstruct *arg1, *arg2;
    pthread_t thread1;

    if (args->size < 20) { // small array
        qsort(args->array, args->size, sizeof(int), intcompare);
        return NULL;
    }

    arg1 = (struct argstruct *)calloc(1, sizeof(struct argstruct));
    arg2 = (struct argstruct *)calloc(1, sizeof(struct argstruct));
    arg1->size = args->size / 2; arg1->array = args->array;
    arg2->size = args->size / 2; arg2->array = &A[args->array/2];

    pthread_create(&thread1, NULL, do_work, arg1);
    do_work((void *)arg2);
    pthread_join(thread1, NULL);
    merge(args->array, args->size);
    return NULL;
}
```

## Recursive Sorting

- Code for the master thread

```
struct {
    int *array;
    int size;
} argstruct;

int main() {
    int *A; // Array of size N
    struct argstruct *arg;
    pthread_t thread1;

    arg = (struct argstruct *)calloc(1, sizeof(struct argstruct));
    arg->size = N; arg->array = A;

    pthread_create(&thread1, NULL, do_work, arg);
    pthread_join(thread1, NULL);

    free(arg);
}
```

## Performance?

- More threads is good
  - The more threads the more parallelism, which is good when we have many cores/processors
- More threads is bad:
  - The more threads the more merging operations
    - But it's hopefully in parallel
  - The more threads the more "thread overhead"
- What about Load Balancing?
  - It is possible that the left branch of the tree, i.e., the left half of the array is more difficult to sort than the right half
  - But since many threads are created recursively, as long as we have P threads we can keep a P-core machine busy
  - Therefore more threads is good:
    - better load balance
- The number of threads is controlled by the depth of the tree, and in our case by the "small array threshold"
- Lesson:** there is probably an optimal "small array threshold" value, which should be determined experimentally

## Domain Decomposition

- Refers to parallelization by assigning pieces of a "domain" to different threads at the onset of the execution
  - each thread operates on a sub-domain throughout execution
  - threads may need to "communicate" with other threads
  - different from (recursive) master-worker execution
- Let's look at a series of examples

## The simple case

- The domain can be decomposed into **equal and independent** parts
- Example: Compute all values of a function of two variables over a 2-D domain
  - function  $f(x,y)$  = <requires many flops>
  - domain =  $([0,10],[0,10])$
  - domain resolution = 0.001
  - number of points =  $(10 / 0.001)^2 = 10^8$
  - number of processors = 10
  - number of threads = 10
  - each thread performs  $10^7$  function evaluations
- No need for thread synchronization
- No need for critical sections

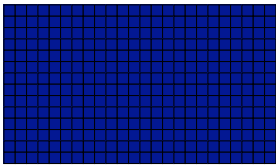


## Dependent Computations

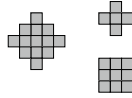
- In many applications, things are not so simple
  - computations may not be independent
  - computations may not be identical or even predictable
- Common example: apply a "stencil" to a 2-D domain
  - Consider a 2-D domain that consists of cells
  - Update every cell of the domain based on neighboring cells
  - Repeat for several iterations
  - Commonly referred to as "stencil applications"
  - CFD, game of life, image processing, etc.

## Stencil Applications

2-D domain



Example stencil shapes



$$d[i][j] = (d[i][j] + d[i-1][j] + d[i+1][j] + d[i][j-1] + d[i][j+1]) / 5$$

$$d[i][j] = (d[i][j] + d[i+1][j] + d[i][j-1] + d[i][j+1]) / 4$$

$$d[i][j] = (d[i][j] + d[i-1][j] + d[i+1][j] + d[i][j+1]) / 4$$

$$d[i][j] = (d[i][j] + d[i-1][j] + d[i][j-1] + d[i][j+1]) / 4$$

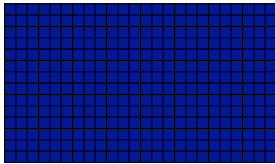
$$d[i][j] = (d[i][j] + d[i-1][j] + d[i+1][j] + d[i][j-1]) / 4$$

## Stencil Applications

- How can we do a stencil application in parallel with threads and shared memory?
  - Identify all the computation that can be done in parallel
  - Identify where synchronization is needed
- In this case it's rather simple
  - All computations independent, synchronization between iterations
  - Have two arrays for the domain
    - one for iteration t
    - one for iteration t+1
  - Algorithm
    - Compute the domain for iteration t+1 using values from iteration t, in a multi-threaded fashion
    - Synchronize all threads with a barrier
    - "Swap" arrays
    - Repeat until maximum number of iterations is reached

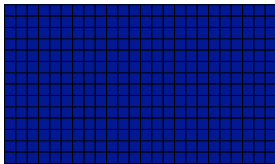
## Stencil Applications

D1



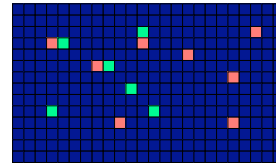
Create two domains  
(i.e., two arrays in memory)

D2



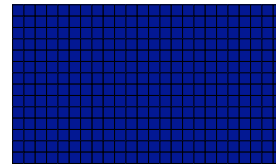
## Stencil Applications

D1



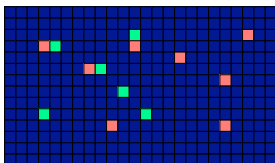
Store some initial values in D1

D2



## Stencil Applications

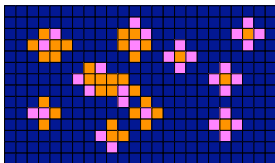
D1



Compute values in D2 based on values in D1

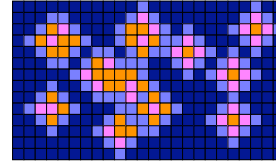
D2 holds values at iteration 1

D2



## Stencil Applications

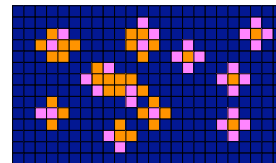
D1



Compute values in D1 based on values in D2

D1 holds values at iteration 2

D2

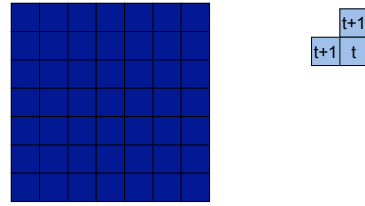


and so on...

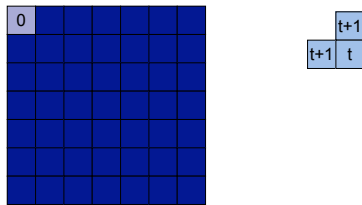
## Stencil applications

- So far we have looked at stencils that use only values obtained at iteration  $t$  for computing values at iteration  $t+1$
- Not all stencils are like this
- Example
$$d[i][j](t+1) = (d[i][j](t) + d[i-1][j](t+1) + d[i][j-1](t+1)) / 3$$
- In this case, not all computations are independent and synchronization becomes more of an issue
- We can implement all this with a single domain

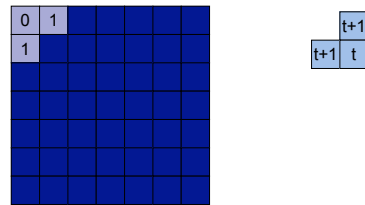
## Stencil applications



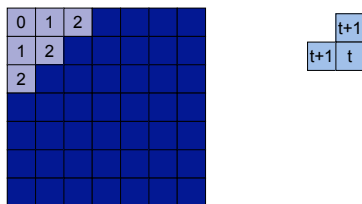
## Stencil applications



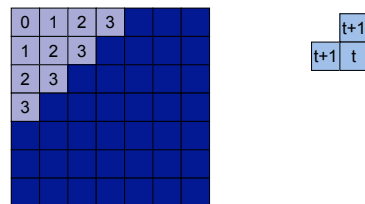
## Stencil applications



## Stencil applications



## Stencil applications



## Stencil applications

0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12



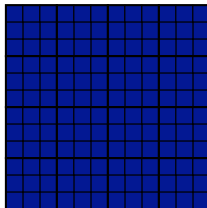
Called a "wavefront" computation

## Wavefront computation

- How can we parallelize a wavefront computation?
- We have seen that the computation consists in computing  $2n-1$  antidiagonals, in sequence.
- Computations within each antidiagonal are independent, and can be done in a multithreaded fashion
- Algorithm:
  - for each antidiagonal
  - use multiple threads to compute its elements
    - one may need to use a variable number of threads because some antidiagonals are very small, while some can be large
  - can be implemented with a single array

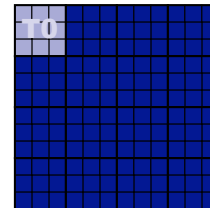
## Wavefront computation

- What about cache efficiency?
- After all, reading only one element from a diagonal at a time is probably not good
- Solution: blocking
  - Just like matrix multiply



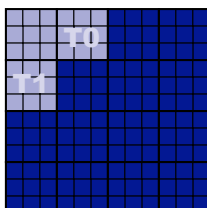
## Wavefront computation

- What about cache efficiency?
- After all, reading only one element from a diagonal at a time is probably not good
- Solution: blocking



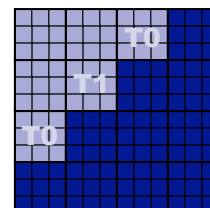
## Wavefront computation

- What about cache efficiency?
- After all, reading only one element from a diagonal at a time is probably not good
- Solution: blocking



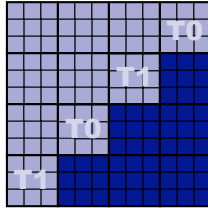
## Wavefront computation

- What about cache efficiency?
- After all, reading only one element from a diagonal at a time is probably not good
- Solution: blocking



## Wavefront computation

- What about cache efficiency?
- After all, reading only one element from a diagonal at a time is probably not good
- Solution: blocking

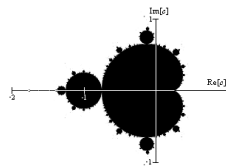


## Load Balancing

- In all the previous examples, all individual computations were identical
- There was no real load balancing issues
  - Although where we compute a diagonal with 10 elements with 4 threads, two of the threads do more work than the other two
  - But there is no real solution to this
  - Tricks could be played in a blocked implementation
    - threads could share blocks
    - not clear what the impact would be on cache utilization

## Load Balancing

- In all the previous examples, all individual computations are identical
- This is not always the case
- Example: Mandelbrot
  - For each complex number  $c$
  - Define the series
    - $Z_0 = 0$
    - $Z_{n+1} = Z_n^2 + c$
  - If the series converges, put a point  $c$
  - If one partitions the domain in 4 squares among 4 threads, some of the threads will have much more work to do than others



## Mandelbrot and Load Balancing

- The problem with the Mandelbrot application is that not all points are created equal
  - Black points: I need to make sure that the series converges, so I just keep going for a long time until I decide that it won't diverge
    - Can take a while
  - White points: the series may diverge immediately
    - Can take no time at all
- So threads/cores that get a lot of black points will compute slower

## Mandelbrot

- Solution: do not partition the computation in as many tiles as threads
    - partition it in many small tiles
- 
- Do the computation in a master-worker fashion
    - threads "request" work only when they have nothing to do
    - as opposed to having a predefined set of things to do
  - Trade-off
    - Small tiles: better load balancing, higher overhead
    - Big tiles: lower overhead, worse load balancing

## Conclusion

- There are many concerns to achieve parallel efficiency
  - Exploit parallelism
    - more tasks is better when more processors/cores
  - Load balancing
    - more tasks is better
  - Overhead
    - fewer tasks is probably better
- In the end, you have to hope you fall in one of the "known" cases, but you often have to resort to your own ingenuity