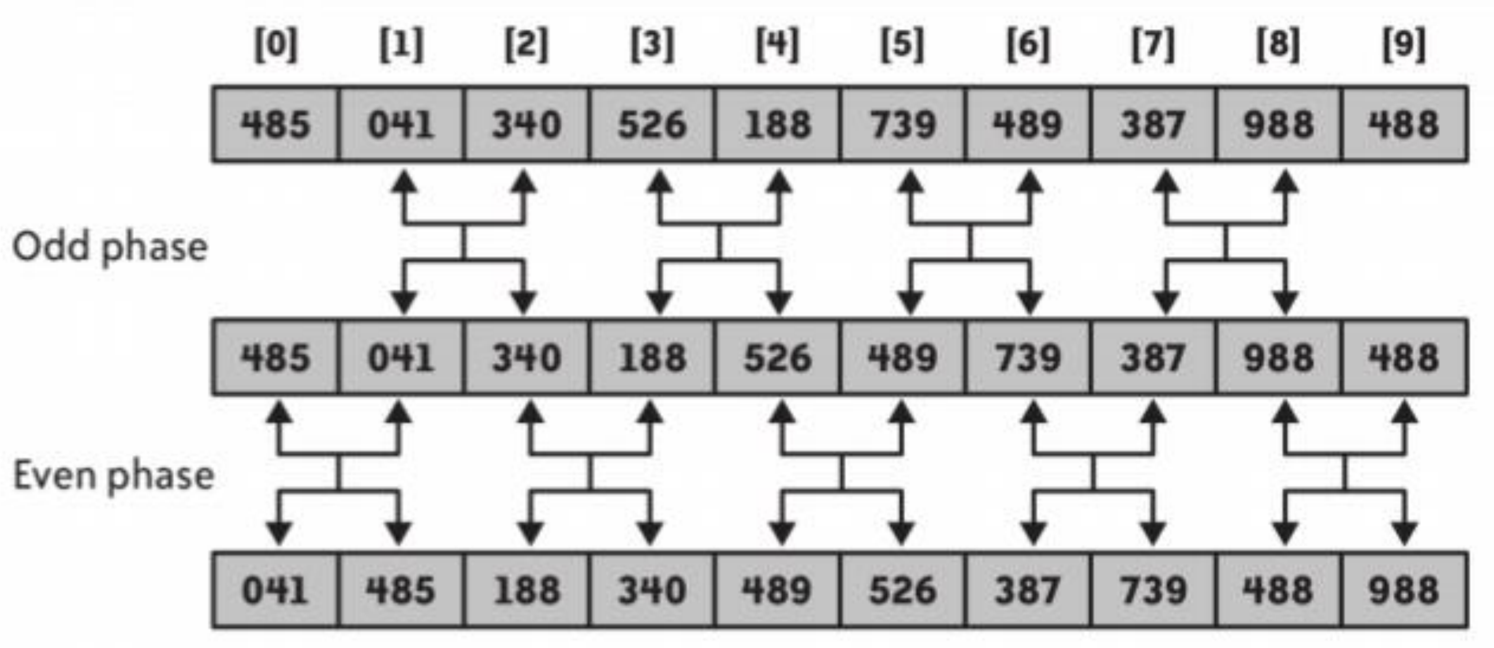
OpenMP Lecture 2

OPENMP EXAMPLES

OpenMP Example – Odd Even Sort

- The Odd-even transposition sort
  - We use array of integers, but it can apply to anything
- Like bubble sort compare & swaps adjacent items
- Unlike bubble sort compares disjointed pairs
- Odd and Even phases are repeated until no further swap is detected



Serial Code

```
void OddEvenSort(int *A, int N)
{
  int exch = 1, start = 0, i;
  int temp;

  while (exch || start) {
    exch = 0;
    for (i = start; i < N-1; i += 2) {
      if (A[i] > A[i+1]) {
        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
        exch = 1;
      }
    }
    if (start == 0) start = 1;
    else start = 0;
  }
}
```

Must go through at least one odd and one even phase

- Hence the start variable

Parallelization using Data Decomposition

```
void OddEvenSort(int *A, int N)
{
  int exch = 1, start = 0, i;
  int temp;

  while (exch || start) {
    exch = 0;

#pragma omp parallel for private(temp)
    for (i = start; i < N-1; i += 2) {
      if (A[i] > A[i+1]) {
        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
        exch = 1;
      }
    }
  if (start == 0) start = 1;
  else start = 0;
  }
}
```

Issues

- *start* does not need protection
  - Read within parallel, but updated outside of it
- *exch* is the opposite: still no need for protection
  - Each thread updates with the same value (benign data race)
- Overhead
  - Each *while* iteration needs starting and stopping threads

Version 2

```
#pragma omp parallel
{
   int temp;
   while (exch || start) {
     exch = 0;
#pragma omp for
     for (i = start; i < N; i += 2){
       if (A[i] > A[i+1]) {
         temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
         exch = 1;
       }
     }
#pragma omp single
     if (start == 0) start = 1;
     else start = 0;
   }
}
```

Race Condition?

- exch may cause race:
  - What if one thread makes exch = 0? Other threads will not enter (exit)!
- Protect using critical?
  - Not enough! Need to make exch the way to let all threads in!

```
#pragma omp parallel
{
  int temp;
  while (1) {
    if (exch == 0 && start == 0) break;
#pragma omp critical
    exch--;
#pragma omp for
    for (i = start; i < N; i += 2) {
      if (A[i] > A[i+1]) {
        temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
#pragma omp critical
        exch = omp_num_threads(); // KEY!
      }
    }
#pragma omp single
    if (start == 0) start = 1;
    else start = 0;
  }
}
```

Still have race issues!

- Why use infinite while(1) loop?
  - Cannot protect read access in while conditional.
  - No need to protect the if conditional!
- Good protection:
  - Decrement and update are protected with the same critical region
- Still have a race:
  - What if a thread enters the for and does a swap and updates exch, while some others still waiting for exch--?
- Solution: need a barrier
- WOW, too many synchronizations!!! Let's go back to old way!

Final Version

```
void OddEvenSort(int *A, int N)
{
  int exch0, exch1 = 1, trips = 0, i;

  while (exch1) {
    exch0 = 0;
    exch1 = 0;

#pragma omp parallel
    {
      int temp;
#pragma omp for
      for (i = 0; i < N-1; i += 2) {
        if (A[i] > A[i+1]) {
          temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
          exch0 = 1;
        }
      }

      if (exch0 || !trips) {
#pragma omp for
        for (i = 1; i < N-1; i += 2) {
          if (A[i] > A[i+1]) {
            temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
            exch1 = 1;
          }
        }
      } // if exch0
    } // end parallel
    trips = 1;
  }
}
```

Prime Number Count Example (page 1)

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>
int prime_number ( int n );
int main ( int argc, char *argv[] ){
  double wtime;
  int n = 1000;
  printf ( " Processors available = %d\n", omp_get_num_procs ( ));
  printf ( " Number of threads =  %d\n", omp_get_max_threads ( ));
  wtime = omp_get_wtime ( );
  primes = prime_number ( n );
  wtime = omp_get_wtime ( ) - wtime;

  return 0;
}
```

Prime Number Count Example (page 2)

```
int prime_number ( int n ){
  int i,j,prime,total;
 # pragma omp parallel shared ( n ) private ( i, j, prime )
   # pragma omp for reduction ( + : total )
  for ( i = 2; i <= n; i++ ){
    prime = 1;
    for ( j = 2; j < i; j++ ){
      if ( i % j == 0 ){
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  return total;
}
```

Summary of OpenMP Lectures

- OpenMP, data-parallel constructs only
  - Task-parallel constructs later
- What's good?
  - Small changes are required to produce a parallel program from sequential (parallel formulation)
  - Avoid having to express low-level mapping details
  - Portable and scalable, correct on 1 processor
- What is missing?
  - Not completely natural if want to write a parallel code from scratch
  - Not always possible to express certain common parallel constructs
  - Locality management
  - Control of performance