Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the `sendrecv` method of MPI

2. The Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the `pthread_barrier_t`

MCS 572 Lecture 30
Introduction to Supercomputing
Jan Verschelde, 29 March 2021
Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the sendrecv method of MPI

2. the Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the pthread_barrier_t
the linear barrier

A barrier has two phases:
1. the arrival or trapping phase;
2. the departure or release phase.

The manager maintains a counter: only when all workers have sent to the manager, does the manager send messages to all workers.

The counter implementation of a barrier or linear barrier is effective but it takes $O(p)$ steps.

<table>
<thead>
<tr>
<th>manager</th>
<th>worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $i$ from 1 to $p - 1$ do</td>
<td></td>
</tr>
<tr>
<td>receive from $i$</td>
<td>send to manager</td>
</tr>
<tr>
<td>for $i$ from 1 to $p - 1$ do</td>
<td></td>
</tr>
<tr>
<td>send to $i$</td>
<td>receive from manager</td>
</tr>
</tbody>
</table>
the linear barrier for $p = 8$
Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the `sendrecv` method of MPI

2. the Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the `pthread_barrier_t`
the tree barrier for $p = 8$
implementing a tree barrier

The trapping phase, for $p = 2^k$ (recall the fan in gather):

for $i$ from $k - 1$ down to 0 do
    for $j$ from $2^i$ to $2^{i+1}$ do
        node $j$ sends to node $j - 2^i$;
        node $j - 2^i$ receives from node $j$.

The release phase, for $p = 2^k$ (recall the fan out scatter):

for $i$ from 0 to $k - 1$ do
    for $j$ from 0 to $2^i - 1$ do
        node $j$ sends to $j + 2^i$;
        node $j + 2^i$ receives from node $j$.

The tree barrier needs $2 \log_2(p)$ stages.

Number of messages: $2 \sum_{i=0}^{k-1} 2^i = 2 \left( \frac{2^k - 1}{2 - 1} \right) = 2^{k+1} - 2 = 2p - 2.$
Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the `sendrecv` method of MPI

2. the Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the `pthread_barrier_t`
the butterfly barrier for $p = 8$

Two processors can synchronize in one step:

Applied to $p = 4$ and $p = 8$, observe there are no idle processors:
the algorithm for a butterfly barrier, for $p = 2^k$

for $i$ from 0 to $k - 1$ do
  $s := 0$;
  for $j$ from 0 to $p - 1$ do
    if $(j \mod 2^{i+1} = 0)$ $s := j$;
    node $j$ sends to node $((j + 2^i) \mod 2^{i+1}) + s$;
    node $((j + 2^i) \mod 2^{i+1}) + s$ receives from node $j$. 

![Diagram of butterfly barrier algorithm]

- Introduction to Supercomputing (MCS 572)
- Barriers for Synchronizations
Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the `sendrecv` method of MPI

2. the Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the `pthread_barrier_t`
avoiding deadlock with sendrecv

\[ P_{i-1} \quad P_i \quad P_{i+1} \]

recv\( (P_i) \quad \rightarrow \quad \text{send}(P_{i-1}) \]
\[ \text{send}(P_i) \quad \rightarrow \quad \text{recv}(P_{i-1}) \]
\[ \text{recv}(P_{i+1}) \quad \leftarrow \quad \text{send}(P_i) \]

is equivalent to

\[ P_{i-1} \quad P_i \quad P_{i+1} \]

sendrecv\( (P_i) \quad \leftrightarrow \quad \text{sendrecv}(P_{i-1}) \]
\[ \text{sendrecv}(P_{i+1}) \quad \leftrightarrow \quad \text{sendrecv}(P_i) \]
the `sendrecv` in MPI

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

where the parameters are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>initial address of send buffer</td>
</tr>
<tr>
<td>sendcount</td>
<td>number of elements in send buffer</td>
</tr>
<tr>
<td>sendtype</td>
<td>type of elements in send buffer</td>
</tr>
<tr>
<td>dest</td>
<td>rank of destination</td>
</tr>
<tr>
<td>sendtag</td>
<td>send tag</td>
</tr>
<tr>
<td>recvbuf</td>
<td>initial address of receive buffer</td>
</tr>
<tr>
<td>recvcount</td>
<td>number of elements in receive buffer</td>
</tr>
<tr>
<td>recvtype</td>
<td>type of elements in receive buffer</td>
</tr>
<tr>
<td>source</td>
<td>rank of source or <code>MPI_ANY_SOURCE</code></td>
</tr>
<tr>
<td>recvtag</td>
<td>receive tag or <code>MPI_ANY_TAG</code></td>
</tr>
<tr>
<td>comm</td>
<td>communicator</td>
</tr>
<tr>
<td>status</td>
<td>status object</td>
</tr>
</tbody>
</table>
a simple illustration

We use **MPI_Sendrecv** to synchronize two nodes:

```
$ mpirun -np 2 ./use_sendrecv
Node 0 will send a to 1
Node 0 received b from 1
Node 1 will send b to 0
Node 1 received a from 0
$
```
using MPI_Sendrecv

#include <stdio.h>
#include <mpi.h>

#define sendtag 100

int main ( int argc, char *argv[] )
{
    int i,j;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&i);

    j = (i+1) % 2; /* the other node */
a bidirectional data transfer

Processors 0 and 1 swap characters:

```c
{  char c = 'a' + (char)i; /* send buffer */  
   printf("Node %d will send %c to %d\n",i,c,j);  
   char d; /* receive buffer */

   MPI_Sendrecv(&c,1,MPI_CHAR,j,sendtag,  
                 &d,1,MPI_CHAR,MPI_ANY_SOURCE,  
                 MPI_ANY_TAG,MPI_COMM_WORLD,&status);

   printf("Node %d received %c from %d\n",i,d,j);
}

MPI_Finalize();
return 0;
```
Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the `sendrecv` method of MPI

2. the Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the `pthread_barrier_t`
A data parallel computation is a computation where the *same* operations are performed on *different* data *simultaneously*.

**Benefits:**
- easy to program,
- scales well,
- fit for SIMD computers.

**Problem:** compute $\sum_{i=0}^{n-1} a_i$ for $n = p = 2^k$.

**Related problem:** composite trapezoidal rule.
the prefix sum for $n = p = 8$

- **Step 1**
  - $a_0$ to $a_7$
  - Summation from 0 to 7

- **Step 2**
  - Summation from 0 to 6

- **Step 3**
  - Summation from 0 to 0
the prefix sum algorithm

For \( n = p = 2^k \), processor \( i \) executes:

\[
\begin{align*}
    s & := 1; x := a_i; \\
    \text{for } j \text{ from 0 to } k - 1 \text{ do} & \\
    & \quad \text{if } (j < p - s + 1) \text{ send } x \text{ to processor } i + s; \\
    & \quad \text{if } (j > s - 1) \text{ receive } y \text{ from processor } i - s; \\
    & \quad \text{add } y \text{ to } x: \ x := x + y; \\
    s & := 2 \times s.
\end{align*}
\]

The speedup: \( \frac{p}{\log_2(p)} \).

Communication overhead: one send/recv in every step.
Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the `sendrecv` method of MPI

2. the Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the `pthread_barrier_t`
#include <stdio.h>
#include "mpi.h"
#define tag 100     /* tag for send/recv */

int main ( int argc, char *argv[] )
{
    int i,j,nb,b,s;
    MPI_Status status;
    const int p = 8;      /* run for 8 processors */

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&i);

    nb = i+1;        /* node i holds number i+1 */
    s = 1;          /* shift s will double in every step */
the prefix sum loop

for(j=0; j<3; j++) /* 3 stages, as log2(8) = 3 */
{
    if(i < p - s) /* every one sends, except last s ones */
        MPI_Send(&nb,1,MPI_INT,i+s,tag,MPI_COMM_WORLD);
    if(i >= s) /* every one receives, except first s ones */
    {
        MPI_Recv(&b,1,MPI_INT,i-s,tag,MPI_COMM_WORLD,&status);
        nb += b; /* add received value to current number */
    }
    MPI_Barrier(MPI_COMM_WORLD); /* synchronize computations */
    if(i < s)
        printf("At step %d, node %d has number %d.\n",j+1,i,nb);
    else
        printf("At step %d, Node %d has number %d = %d + %d.\n",j+1,i,nb,nb-b,b);
    s *= 2; /* double the shift */
}
if(i == p-1) printf("The total sum is %d.\n",nb);
running the code

$ mpirun -np 8 ./prefix_sum
At step 1, node 0 has number 1.
At step 1, Node 1 has number 3 = 2 + 1.
At step 1, Node 2 has number 5 = 3 + 2.
At step 1, Node 3 has number 7 = 4 + 3.
At step 1, Node 7 has number 15 = 8 + 7.
At step 1, Node 4 has number 9 = 5 + 4.
At step 1, Node 5 has number 11 = 6 + 5.
At step 1, Node 6 has number 13 = 7 + 6.
At step 2, node 0 has number 1.
At step 2, node 1 has number 3.
At step 2, Node 2 has number 6 = 5 + 1.
At step 2, Node 3 has number 10 = 7 + 3.
At step 2, Node 4 has number 14 = 9 + 5.
At step 2, Node 5 has number 18 = 11 + 7.
At step 2, Node 6 has number 22 = 13 + 9.
At step 2, Node 7 has number 26 = 15 + 11.
At step 3, node 0 has number 1.
At step 3, node 1 has number 3.
At step 3, node 2 has number 6.
At step 3, node 3 has number 10.
At step 3, Node 4 has number $15 = 14 + 1$.
At step 3, Node 5 has number $21 = 18 + 3$.
At step 3, Node 6 has number $28 = 22 + 6$.
At step 3, Node 7 has number $36 = 26 + 10$.
The total sum is 36.
Barriers for Synchronizations

1. Synchronizing Computations
   - the linear barrier
   - the tree barrier
   - the butterfly barrier
   - the sendrecv method of MPI

2. the Prefix Sum Algorithm
   - data parallel computations
   - the prefix sum algorithm in MPI

3. Barriers in Shared Memory Parallel Programming
   - an example illustrating the pthread_barrier_t
Recall Pthreads and the work crew model. Often all threads must wait till on each other.

```c
int count = 3;
pthread_barrier_t our_barrier;
p_thread_barrier_init(&our_barrier, NULL, count);
```

In the example above, we initialized the barrier that will cause as many threads as the value of `count` to wait.

A thread remains trapped waiting as long as fewer than `count` many threads have reached `pthread_barrier_wait(&our_barrier);`

and the `pthread_barrier_destroy(&our_barrier)` should only be executed after all threads have finished.
running an illustrative program

The shared data is the time each thread sleeps.

```
$ ./pthread_barrier_example
Give the number of threads : 5
Created 5 threads ...
Thread 0 has slept 2 seconds ...
Thread 2 has slept 2 seconds ...
Thread 1 has slept 4 seconds ...
Thread 3 has slept 5 seconds ...
Thread 4 has slept 6 seconds ...
Thread 4 has data : 24256
Thread 3 has data : 24256
Thread 2 has data : 24256
Thread 1 has data : 24256
Thread 0 has data : 24256
$
```

Each thread prints only after all data is ready.
headers and global variables

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

int size;    /* size equals the number of threads */
int *data;   /* shared data, as many ints as size */
pthread_barrier_t our_barrier; /* to synchronize */
```

The global variables will be initialized in the main program:

- the user is prompted to enter `size`, the number of threads;
- the array `data` is allocated with `size` elements;
- the barrier `our_barrier` is initialized.
**code executed by each thread**

```c
void *fun ( void *args )
{
    int *id = (int*) args;
    int r = 1 + (rand() % 6);
    int k;
    char strd[size+1];

    sleep(r);
    printf("Thread %d has slept %d seconds ...\n", *id, r);
    data[*id] = r;

    pthread_barrier_wait(&our_barrier);

    for(k=0; k<size; k++) strd[k] = '0' + ((char) data[k]);
    strd[size] = '\0';

    printf("Thread %d has data : %s\n", *id, strd);
}
```
the main function

```c
int main ( int argc, char* argv[] )
{
    printf("Give the number of threads : "); scanf("%d", &size);
data = (int*) calloc(size, sizeof(int));
{
    pthread_t t[size];
    pthread_attr_t a;
    int id[size], i;

    pthread_barrier_init(&our_barrier, NULL, size);

    for(i=0; i<size; i++)
    {
        id[i] = i;
        pthread_attr_init(&a);
        if(pthread_create(&t[i], &a, fun, (void*)&id[i]) != 0)
            printf("Unable to create thread %d!\n", i);
    }
    printf("Created %d threads ...
", size);
    for(i=0; i<size; i++) pthread_join(t[i], NULL);

    pthread_barrier_destroy(&our_barrier);
}
return 0;
}```
Summary + Exercises

We started chapter 6 in the book of Wilkinson and Allen.

Exercises:

1. Write code using `MPI_sendrecv` for a butterfly barrier. Show that your code works for $p = 8$.

2. Rewrite `prefix_sum.c` using `MPI_sendrecv`.

3. Consider the composite trapezoidal rule for the approximation of $\pi$, doubling the number of intervals in each step. Can you apply the prefix sum algorithm so that at the end, processor $i$ holds the approximation for $\pi$ with $2^i$ intervals?