Thinking parallel

- The following computes the sum of $x[0] + ... + x[15]$ serially:

  ```
  For (i = 1 ; i < 16 ; i++)
  {
    x[0] = x[0] + x[i]
  }
  ```

- Takes $n-1$ steps to sum $n$ numbers on one processor

- Applies to associative and commutative operations (+, *, min, max, …)

Parallel sum algorithm (on 8 processors)

- Takes $\log n$ steps to sum $n$ numbers on $n/2$ processor

```plaintext
P0

P0

P0

x[0]=

x[0]= x[0]+x[1] =1+2 =3


X[0]=36+100

X[0]=10+26

X[0]=3+7

X[4]=11+15

X[8]=42+58

X[8]=19+23

X[12]=27+31

```
Example code on SMP

```c
half = 8;
repeat
{  half = half/2;
    if (Pid < half) x[Pid] = x[Pid] + x[Pid+half];  }
until (half == 1);
```

Potential for race conditions??

Processor 1  Processor 2  Processor 3

Barrier synchronization

Should “half” be private or shared?

Pid is the processor ID

Example: when \( p = 10 \) (not a power of 2)

```c
half = 10;
repeat
{
    if (half % 2 != 0 && Pid == 0) /* when half is odd; P0 gets the last element */
        x[0] = x[0] + x[half-1];
    half = half/2;
    if (Pid < half) x[Pid] = x[Pid] + x[Pid+half];
    barrier synch();
}
until (half == 1);
```

Now, we want to sum \( n \) elements on \( p \) processors, \( n >> p \)
Parallel sum of 16 elements on 4 processors

• Divide the array to be summed into 4 parts and assign one part to each processor

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
</table>

• Need 5 steps to sum 16 numbers on 4 processors
  ▪ Speedup = 15/5 = 3

• Need 255+2 steps to sum 1024 numbers on 4 processors
  ▪ Speedup = 1023/257 = 3.9

• How long does it take to sum \( n \) numbers on \( p \) processors?
  ▪ Speedup = ??

Parallel sum on a shared address space machine

• Assume \( x[0] \ldots x[9999] \) are stored in shared memory.
• Assume \( P = 16 \) processors, each with an identifier \( Pid \) (between 0 and 15)
• To sum the 10000 numbers, each processor executes the following:

```c
sum[Pid] = 0;
for (i = 625 * Pid ; i < 625 * (Pid +1) ; i++)
    sum[Pid] = sum[Pid] + x[i];
half= 8 ; /* P = 16 */
for (i=0 ; i < 4 ; i++)
    { synchronize ; /* a barrier */
    if(Pid < half ) sum[Pid] = sum[Pid] + sum[Pid + half ];
    half = half / 2; }
```

• \( sum[] \) and \( x[] \) are shared arrays.
• \( half, Pn \) and \( i \) are private variables (each processor has its own copy).
• Where will the global sum end up being?
• What if we want all processors to get a copy of the global sum?
• How would you change the program if \( P \) is not a power of two?
• Rewrite the program in terms of the \# of processors and the size of \( x \)?
EX: Computing the dot product on shared memory

Example: dot product of two vectors, \(x\) and \(y\) (using a single thread)

\[
dp = 0; \\
for (i = 0 ; i < n ; i++) \\
dp += x[i] * y[i]
\]

Using 4 processors:
- Partition the arrays into 4 parts
- Each processor computes a partial sum
- One processor sums up the partial sums (could use tree binary reduction)

Multi-thread version of the dot product example

- Multi-threading was originally designed for Hiding Memory Latency
- With multi cores, multiple threads will execute on multiple cores

\[
dp = 0; \\
for (k = 0; k < 4; k++) /* fork 4 threads */ \\
create_thread (partial_product, k, n); /* k is used as a thread id */
\]

Wait until all threads return; /* join threads */

\[
for (k = 0; k < 4; k++) \\
dp += pdp[k];
\]

void partial_product (k, n);
\{ \\
\}
Another version of the dot product example

```c
dp = 0;
for (k = 0; k < 4; k++)
    create_thread(partial_product, k, n);
Wait until all threads return;

void partial_product(k, n);
{
    int pdp = 0; /* each thread has its own copy of the local variable pdp */
    for (i = k*n/4; i < (k+1)*n/4; i++)
        pdp += x[i] * y[i];
    pd += pdp;
    return;
}
```

Synchronization (race conditions)

What is the output of the following program??

```c
dp = 0;
for (id = 0; id < 4; id++)
    create_thread(…, count, …);

void count()
{
    dp = dp + 1;
}
```

- A critical section is a section of code that can be executed by one processor at a time (to guarantee mutual exclusion)
- `locks` can be used to enforce mutual exclusion

Most parallel languages provides ways to declare and use locks or critical sections
Mutual Exclusion

• We need mutual exclusion in both parallel and serial programs (why?)

• Locks can be used to allow mutual exclusion, and hence provide a mechanism for exclusive access to shared data.

• Hardware support (in the form of atomic operations) is needed to implement locks
  – Atomic load-modify-store instructions,
  – Atomic swap instructions (swap the contents of a memory location with that of a register).

• In cache coherent systems, a memory location should be in the “Exclusive” state while executing an atomic operation on this location.

Implementing locks using atomic swap

• Atomic Swap interchanges a value in a register for a value in memory
  • loads the value from a memory location into the register
  • stores the value in register into the memory location

• Atomic swap can be used to implement locks:
  • The lock is represented by a variable, L
    • L=1 → locked
    • L=0 → not locked

  Lock (L):
  Put 1 in Register, R
  Repeat
    Atomic Swap (R, L)
  Until (R = 0)

  Unlock:
  L = 0
Barrier synchronization

• A barrier synchronization between N threads can be implemented using a shared variable initialized to N.
• When a processor reaches the barrier, it decrements the shared variable by 1 and waits (in a busy wait loop) until the value of the variable is equal to zero before it leaves the barrier.

• Need locks???

• What if there is no shared variables (distributed memory machines)?

• Can you synchronize using special hardware?