Thinking parallel

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

```c
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

![Diagram of parallel sum algorithm on 8 processors]
Example code on SMP

```c
half = 8;
repeat
    half = half/2;
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

Potential for race conditions??

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

```c
half = 10;
repeat
    if (half % 2 != 0 && Pn == 0) /*when half is odd; P0 gets the last element*/
        sum[0] = sum[0] + sum[half-1];
    half = half/2;
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+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
- 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 \( A[0] \ldots A[9999] \) are stored in shared memory.
- Assume \( P = 16 \) processors, each with an identifier \( P_n \) (between 0 and 15)
- To sum the 10000 numbers, each processor executes the following:

```c
Sum[Pn] = 0;
for ( i = 625 * Pn ; i < 625 * (Pn + 1) ; i++)
    Sum[Pn] = Sum[Pn] + A[i];
Half= 8 ; /* P = 16 */
for (i=0 ; i < 4 ; i++)
    { synchronize ; /* a barrier */
    if(Pn < Half ) Sum[Pn] = Sum[Pn] + Sum[Pn + Half ];
    Half = Half / 2; }
```

- \( Sum[ \] \) and \( A[ \] \) are shared arrays,
- \( Half, P_n \) 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 \( A \)?
EX: Computing the dot product on shared memory

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

\[
\begin{align*}
dp &= 0; \\
\text{for} \ (i = 0; \ i < n; \ i++) & \ \\
\quad \ dp += x[i] \times y[i]
\end{align*}
\]

Using 4 processors:
- Partition the arrays into 4 parts
- Each processor computes a partial sum
- One processor sums up the partial sums (or 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

\[
\begin{align*}
dp &= 0; \\
\text{for} \ (k = 0; \ k < 4; \ k++) & \ \\
\quad \ create\_thread\ (\text{partial\_product}, \ k, \ n); \ /* \ k \ is \ used \ as \ a \ thread \ id */ \\
\text{Wait \ until \ all \ threads \ return} & \ \\
\text{for} \ (k = 0; \ k < 4; \ k++) & \ \\
\quad \ dp += \ pdp[k]; \\
\text{void \ partial\_product} \ (k, \ n); \\
\quad \ pdp[k] = 0; \\
\text{for} \ (i = k\times n/4; \ i < (k+1)\times n/4; \ i++) & \ \\
\quad \quad \ pdp[k] += x[i] \times y[i]; \\
\quad \ return;
\end{align*}
\]
Another version of the dot product example

\[
dp = 0 ; \\
for (k = 0; k < 4; k++) \\
    \text{create\_thread(partial\_product, } k, n \text{);} \\
\text{Wait until all threads return ;}
\]

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 ;
}

Shared (global) variables

\[
\begin{array}{c}
\text{x} \\
\text{y} \\
\hline
\text{PE0} \\
\text{PE1} \\
\text{PE2} \\
\text{PE3} \\
\end{array}
\]

\[
\begin{array}{c}
\text{dp} \\
\hline
\end{array}
\]

\(\text{load } \text{dp} \text{ from memory}\)

\(\text{Add } \text{pdp } \text{to } \text{dp}\)

\(\text{store } \text{dp } \text{to memory}\)