Thinking parallel

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

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

- 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

---

\[
\begin{align*}
P0 & \quad x[0]=3+7 \quad x[2]=3+4 \quad x[4]=5+6 \quad x[6]=7+8 \\
P2 & \quad x[1]=10+26 \\
P4 & \quad x[8]=42+58 \\
P6 & \quad x[12]=27+31
\end{align*}
\]
Example code on SMP

half = 4;
repeat
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
  half = half/2;
until (half == 1);

Pn is the processor ID

Potential for race conditions??

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

half = 5;
repeat
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
  /* when half is odd; P0 gets the last element */
  if (half % 2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
  barrier synch();
  half = half/2;
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 $Pn$ (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,
- $Half, Pn$ 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 $A$?
EX: Computing the dot product on shared memory

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

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

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

\[
\begin{align*}
dp &= 0 \\
for (k = 0; k < 4; k++) \quad /* fork 4 threads */ \\
&\quad create_thread (partial_product, k*n/4, n/4); \\
Wait until all threads return; \quad /* join threads */ \\
for (k = 0; k < 4; k++) \\
&\quad dp += pdp[k] \\
\end{align*}
\]

\[
\begin{align*}
void partial_product (a , b); \\
\{ \\
&\quad pdp[k] = 0 \\
for (i = a ; \ i < a+b ; \ i++) \\
&\quad \quad pdp[k] += x[i] * y[i] ; \\
\&\quad return ; \\
\}
\end{align*}
\]

Shared (global) variables

\[
\begin{align*}
x & \quad y \\
pdp & \quad dp \\
\end{align*}
\]
P threads (POSIX)

- A thread is a lightweight process (has its own stack and execution state, but shares the address space with its parent).
- Hence, threads have local data but also can share global data.

The Pthread API

(see https://computing.llnl.gov/tutorials/pthreads/)

- Pthreads has emerged as the standard threads API (Application Programming Interface), supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
- Provides two basic functions for specifying concurrency:

  ```c
  #include <pthread.h>

  int pthread_create (pthread_t *thread_handle,
                     const pthread_attr_t *attribute,
                     void (*thread_function)(void *),
                     void *arg);

  int pthread_join (pthread_t thread_handle,
                    void * *ptr);
  ```
Another version of the dot product example

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

void partial_product (a, b);
{ int pdp = 0; /* each thread has its own copy of pdp */
  for (i = a; i < a+b; i++)
      pdp += x[i] * y[i];
  pd += pdp;
  return;
}
```

Shared (global) variables

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
void partial_product (a, b);
{ int pdp = 0; /* each thread has its own copy of pdp */
  for (i = a; i < a+b; i++)
      pdp += x[i] * y[i];
  pd += pdp;
  return;
}
```