#include <omp.h>

// partitions s..e into nc chunks, placing the
// ith in first and last
void chunker(int s, int e, int nc, int i,  
{ int chunksize = (e-s+1) / nc;  
  first = s + i * chunksize;  
  last = first + chunksize - 1;  
  else last = e;  
} 

// returns the "dot product" of vectors u and v  
float innerprod(float *u, float *v, int n)  
{ float sum = 0.0; int i;  
  for (i = 0; i < n; i++)  
    sum += u[i] * v[i];  
  return sum;  
} 

#include <cuda.h>

// compare and swap, thread number "me"; copies from f to t,  
// swapping f[i] and f[j] if the higher-index value is smaller;  
// it is required that i < j  
________________cas(int *f, int *t, int i, int j, int n, int me)  
{ if (i < 0 || j >= n) return;  
  if (me == i) {  
    if (f[i] > f[j]) t[me] = f[j];  
    else t[me] = f[i];  
  } else {  
    if (f[i] > f[j]) t[me] = f[i];  
    else t[me] = f[j];  
  }  
}  

// does one iteration of the sort  
__global__ void oekern(int *da, int *daaux, int n, int iter)  
{ int bix = _____________________________;  
  else t[me] = f[i];  
  if (f[i] > f[j]) t[me] = f[j];  
  if (iter % 2) {  
    if (bix % 2) cas(da,daaux,bix-1,bix,n,bix);  
    else cas(da,daaux,bix,bix+1,n,bix);  
  } else {  
    if (bix % 2) cas(da,daaux,bix,bix+1,n,bix);  
    else cas(da,daaux,bix-1,bix,n,bix);  
  }  
}  

// sorts the array ha, length n, using odd/even transp. sort;  
// kept simple for illustration, no optimization  
void oddeven(int *ha, int n)  
{ int mid;  
  if (n <= 1) return;  
  int half = n / 2;  
  oddeven(ha, half);  
  evenodd(ha + half, half);  
  cas(ha, ha + half, half, half, n, half);  
}  

3. A wavefront operation on an nxn matrix A works on diagonals. Here we will define the $i^{th}$ “wave” to consist of $a_{i0}, a_{i-1,1}, ... , a_{0,i}$. (Note that these diagonals are perpendicular to the usual ones.) Suppose we are developing a highly parallel implementation, say in CUDA. Say we set up the calculation for wave $i$ so that a different thread handles each element in the wave. So, assigning from “southwest to northeast,” thread 0 would handle $a_{i0}$, thread 1 would handle $a_{i-1,1}$, and so on.

(a) (15) Fill in the blanks: An obvious problem is that some threads have less work to do than others. In standard terminology, we say that this is a ______________ problem. Thread $i$ will only be busy a fraction ______________ of the time, $i = 0,1,...,n-1$.

(b) (15) Suppose $n = 7$ and shared memory has 8 banks. For which values of $i$ will the $i^{th}$ wave generate no bank conflicts?
1.

```c
#include <omp.h>

// partitions s..e into nc chunks, placing the ith in first and last (i
// = 0,...,nc-1)
void chunker(int s, int e, int nc, int i, int *first, int *last)
{ int chunksize = (e-s+1) / nc;
  *first = s + i * chunksize;
  if (i < nc-1) *last = *first + chunksize - 1;
  else *last = e;
}
```

```c
// returns the "dot product" of vectors u and v
float innerprod(float *u, float *v, int n)
{ float sum = 0.0;
  for (int i = 0; i < n; i++)
    sum += u[i] * v[i];
  return sum;
}
```

```c
// solves AX = Y, A nxn; stop iteration when total change is < n*eps
void jacobi(float *a, float *x, float *y, int n, float eps)
{
  float *oldx = malloc(n*sizeof(float));
  float se;
  #pragma omp parallel
  { int i;
    int thn = omp_get_thread_num();
    int nth = omp_get_num_threads();
    int first,last;
    chunker(0,n-1,nth,thn,&first,&last);
    for (i = first; i <= last; i++) oldx[i] = x[i] = 1.0;
    float tmp;
    while (1) {
      for (i = first; i <= last; i++) {
        tmp = innerprod(&a[n*i],oldx,n);
        x[i] = (y[i] - tmp) / a[n*i+i];
      }
      #pragma omp barrier
      #pragma omp for reduction(+:se)
      for (i = first; i <= last; i++)
        se += abs(x[i]-oldx[i]);
      #pragma omp barrier
      if (se < n*eps) break;
      for (i = first; i <= last; i++)
        oldx[i] = x[i];
    }
  }
}
```

2. Some students pointed out that swapping `da` and `daaux` was better than copying. This is actually what I intended in the first place, but during debugging I “temporarily” change it to a copy, which was easier. I then forgot to change it back later.

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

// compare and swap; copies from the f to t, swapping f[i] and
// f[j] if the higher-index value is smaller; it is required that i < j
__device__ void cas(int *f,int *t,int i,int j, int n, int me)
{ if (i < 0 || j >= n) return;
  if (me == i) {
    if (f[i] > f[j]) t[me] = f[j];
    else t[me] = f[i];
  } else { // me == j
    if (f[i] > f[j]) t[me] = f[i];
    else t[me] = f[j];
  }
}
```

```c
// does one iteration of the sort
__global__ void oekern(int *da, int *daaux, int n, int iter)
{ int bix = blockIdx.x; // block number within grid
  if (iter % 2) cas(da,daaux,bix-1,bix+1,n,bix);
  else {
    if (bix % 2) cas(da,daaux,bix-1,n,bix);
    else cas(da,daaux,bix,n,bix);
  }
}
```

3. This problem was slightly misspecified, which made it a little easier. There really should be 2n-1 waves, not just n.

(a) This is a load balancing problem. Thread i will be busy only a fraction (n-i)/n of the time.

(b) Without loss of generality, assume that the array starts at word address 0. Note that consecutive elements within a wave are 6 words apart. Then we have the following table:

<table>
<thead>
<tr>
<th>wave</th>
<th>addresses</th>
<th>banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1,7</td>
<td>1,7</td>
</tr>
<tr>
<td>2</td>
<td>2,8,14</td>
<td>2,0,6</td>
</tr>
<tr>
<td>3</td>
<td>3,9,15,21</td>
<td>3,1,7,5</td>
</tr>
<tr>
<td>4</td>
<td>4,10,16,22,28</td>
<td>4,2,0,6,4</td>
</tr>
<tr>
<td>5</td>
<td>5,11,17,23,29,35</td>
<td>5,3,1,7,5,3</td>
</tr>
<tr>
<td>6</td>
<td>6,12,18,24,30,36,42</td>
<td>6,4,2,0,6,4,2</td>
</tr>
</tbody>
</table>

Only threads 0, 1, 2 and 3 avoid bank conflicts.