Name: _____

Directions: **Work only on this sheet** (on both sides, if needed). MAKE SURE TO COPY YOUR ANSWERS TO A SEPARATE SHEET FOR SENDING ME AN ELECTRONIC COPY LATER.

IMPORTANT NOTE: If you believe that nothing needs to be placed into a blank, simply give NA as your answer.

1. (40) You know that array *padding* is used to try to get better parallel access to memory banks. The code below is aimed to provide utilities to assist in this. Details are explained in the comments.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  // routines to initialize, read and write
5  // padded versions of a matrix of floats;
6  // the matrix is nominally mxn, but its
7  // rows will be padded on the right ends,
8  // so as to enable a stride of s down each
9  // column; it is assumed that s >= n
10
11 // allocate space for the padded matrix,
12 // initially empty
13 float *padmalloc(int m, int n, int s) {
14    return malloc(BLANKa);
15 }
16
17 // store the value tostore in the matrix q,
18 // at row i, column j; m, n and
19 // s are as in padmalloc() above
20 void setter(float *q, int m, int n, int s,
21       int i, int j, float tostore) {
22    BLANKb
23 }
24
25 // fetch the value in the matrix q,
26 // at row i, column j; m, n and s are
27 // as in padmalloc() above
28 float getter(float *q, int m, int n, int s,
29       int i, int j) {
30    BLANKc
31 }
32
33 // test example
34 int main() {
35    int i; float *q;
36    q = padmalloc(2,2,3);
37    setter(q,2,2,3,1,0,8);
38    printf("%f\n",getter(q,2,2,3,1,0));
39 }
40
41 // check, using GDB
42 // Breakpoint 1, main () at padding.c:31
43 // 31       printf("%f\n",getter(q,2,2,3,1,0));
44 // (gdb) x/6f q
45 // 0x804b008:      0       0       0       8
46 // 0x804b018:      0       0
```

2. (60) The code below does root-finding. The problem and the strategy used by the code are explained in the comments.

Pointers to functions are used. You probably have seen these before, but if not don't worry about it; it doesn't affect the parts of the code you must fill in. Suffice it to say that the user-supplied function does get called properly.

```
1  #include<omp.h>
2  #include<math.h>
3
4  // OpenMP example:  root finding
5
6  // the function f() is known to be negative
7  // at a, positive at b, and to have exactly
8  // one root in (a,b); the procedure runs
9  // for niters iterations
10
11 // strategy: in each iteration, the current
12 // interval is split into nth equal parts,
13 // and each thread checks its subinterval
14 // for a sign change of f(); if one is
15 // found, this subinterval becomes the
16 // new current interval; the current guess
17 // for the root is the left endpoint of the
18 // current interval
19
20 // of course, this approach is useful in
21 // parallel only if f() is very expensive
22 // to evaluate
23
24 // for simplicity, assumes that no endpoint
25 // of a subinterval will ever exactly
26 // coincide with a root
27
28 float root(float(*f)(float),
29    float inita, float initb, int niters) {
30    BLANKa
31    BLANKb
32    {
33       int nth = omp_get_num_threads();
34       int me = omp_get_thread_num();
35       int iter;
36       BLANKc
37       for (iter = 0; iter < niters; iter++) {
38          BLANKd
39          subintwidth = (currb - curra) / nth;
40          myleft = curra + me * subintwidth;
41          myright = myleft + subintwidth;
42          if ((*f)(myleft) < 0 &&
43                (*f)(myright) > 0) {
44             curra = myleft;
45             currb = myright;
46          }
47       }
48    }
49    return curra;
50 }
51
52 // example
53 float testf(float x) {
54    return pow(x-2.1,3);
55 }
56
57 int main(int argc, char **argv)
58 {  // should print 2.1
59    printf("%f\n",root(testf,-4.1,4.1,1000));
60 }
```

Solutions:

1.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  // routines to initialize, read and write
5  // padded versions of a matrix of floats;
```

```
6  // the matrix is nominally mxn, but its
7  // rows will be padded on the right ends,
8  // so as to enable a stride of s down each
9  // column; it is assumed that s >= n
10
11 // allocate space for the padded matrix,
12 // initially empty
13 float *padmalloc(int m, int n, int s) {
14    return(malloc(m*s*sizeof(float)));
15 }
16
17 // store the value tostore in the matrix q,
18 // at row i, column j; m, n and
19 // s are as in padmalloc() above
20 void setter(float *q, int m, int n, int s,
21       int i, int j, float tostore) {
22    *(q + i*s+j) = tostore;
23 }
24
25 // fetch the value in the matrix q,
26 // at row i, column j; m, n and s are
27 // as in padmalloc() above
28 float getter(float *q, int m, int n, int s,
29       int i, int j) {
30    return *(q + i*s+j);
31 }
32
33 int main() {
34    int i; float *q;
35    q = padmalloc(2,2,3);
36    setter(q,2,2,3,1,0,8);
37    printf("%f\n",getter(q,2,2,3,1,0));
38 }
39
40 // check, using GDB
41 // Breakpoint 1, main () at padding.c:31
42 // 31          printf("%f\n",getter(q,2,2,3,1,0));
43 // (gdb) x/6f q
44 // 0x804b008:      0       0       0       8
45 // 0x804b018:      0       0
```

2.

```
1  #include<omp.h>
2  #include<math.h>
3
4  // OpenMP example:  root finding
5
6  // the function f() is known to be negative
7  // at a, positive at b, and thus has at
8  // least one root in (a,b); if there are
9  // multiple roots, only one is found;
10 // the procedure runs for niters iterations
11
12 // strategy: in each iteration, the current
13 // interval is split into nth equal parts,
14 // and each thread checks its subinterval
15 // for a sign change of f(); if one is
16 // found, this subinterval becomes the
17 // new current interval; the current guess
18 // for the root is the left endpoint of the
19 // current interval
20
21 // of course, this approach is useful in
22 // parallel only if f() is very expensive
23 // to evaluate
24
25 // for simplicity, assumes that no endpoint
26 // of a subinterval will ever exactly
27 // coincide with a root
28
29 float root(float(*f)(float),
30       float inita, float initb, int niters) {
31    float curra = inita;
32    float currb = initb;
33    #pragma omp parallel
34    {
35       int nth = omp_get_num_threads();
36       int me = omp_get_thread_num();
37       int iter;
38       for (iter = 0; iter < niters; iter++) {
39          #pragma omp barrier
40          float subintwidth =
41             (currb - curra) / nth;
42          float myleft =
43             curra + me * subintwidth;
44          float myright = myleft + subintwidth;
45          if ((*f)(myleft) < 0 &&
46              (*f)(myright) > 0) {
47             curra = myleft;
48             currb = myright;
49          }
50       }
51    }
52    return curra;
53 }
54
55 float testf(float x) {
56    return pow(x-2.1,3);
57 }
58
59 int main(int argc, char **argv)
60 {  printf("%f\n",root(testf,-4.1,4.1,1000));
   }
```