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.

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

// routines to initialize, read and write
// padded versions of a matrix of floats;
// the matrix is nominally m x n, but its
// rows will be padded on the right ends,
// so as to enable a stride of s down each
// column; it is assumed that s >= n

// allocate space for the padded matrix,
// initially empty
float *padmalloc(int m, int n, int s) {
    return malloc(BLANKa);
}

// store the value tostore in the matrix q,
// at row i, column j; m, n and
// s are as in padmalloc() above
void setter(float *q, int m, int n, int s,
            int i, int j, float tostore) {
    BLANKb
}

// fetch the value in the matrix q,
// at row i, column j; m, n and s are
// as in padmalloc() above
float getter(float *q, int m, int n, int s,
             int i, int j) {
    BLANKc
}

// test example
int main() {
    int i; float *q;
    q = padmalloc(2,2,3);
    setter(q,2,2,3,1,0,8);
    printf("%f\n",getter(q,2,2,3,1,0));
}

// check, using GDB
// Breakpoint 1, main() at padding.c:31
// 31 printf("%f\n",getter(q,2,2,3,1,0));
// (gdb) x/6f q
// 0x804b008: 0 0 0 0 8
// 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.

```c
#include<omp.h>
#include<math.h>

// OpenMP example: root finding
// the function f() is known to be negative
// at a, positive at b, and to have exactly
// one root in (a,b); the procedure runs
// for niter iterations

// strategy: in each iteration, the current
// interval is split into nth equal parts,
// and each thread checks its subinterval
// for a sign change of f(); if one is
// found, this subinterval becomes the
// new current interval; the current guess
// for the root is the left endpoint of the
// current interval

// of course, this approach is useful in
// parallel only if f() is very expensive
// to evaluate

// for simplicity, assumes that no endpoint
// of a subinterval will ever exactly
// coincide with a root

float root(float(*f)(float),
float inita, float initb, int niter) {
    BLANKa
    BLANKb
    {
        int nth = omp_get_num_threads();
        int me = omp_get_thread_num();
        int iter;
        BLANKc
        for (iter = 0; iter < niter; iter++) {
            BLANKd
            subintwidth = (currb - curra) / nth;
            myleft = curra + me * subintwidth;
            myright = myleft + subintwidth;
            if ((f(myleft) < 0) &&
                (f(myright) > 0) ) {
                currb = myleft;
                curra = myright;
            }
        }
    }
    return curra;
}

// example
float testf(float x) {
    return pow(x-2.1,3);
}

int main(int argc, char **argv) {
    // should print 2.1
    printf("%f\n",root(testf,-4.1,4.1,1000));
}
```

Solutions:

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.

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

// routines to initialize, read and write
// padded versions of a matrix of floats;
```
allocate space for the padded matrix,

initially empty

float *padmalloc(int m, int n, int s) {
    return (malloc(m*s*sizeof(float))); } 

store the value to store in the matrix q,
at row i, column j; m, n and s are as in padmalloc() above

void setter(float *q, int m, int n, int s,
    int i, int j, float tostore) {
    *(q + i*s+j) = tostore; 
}

fetch the value in the matrix q,
at row i, column j; m, n and s are as in padmalloc() above

float getter(float *q, int m, int n, int s,
    int i, int j) {
    return *(q + i*s+j); 
}

int main() {
    int i; float *q;
    q = padmalloc(2,2,3);
    setter(q,2,2,3,1,0,8);
    printf("%f
",getter(q,2,2,3,1,0)); 
}

check, using GDB
// Breakpoint 1, main () at padding.c:31
// 31 printf("%f\n",getter(q,2,2,3,1,0));
(gdb) x/6f q
0x804b008: 0 0 0 0 0
0x804b018: 0 0

2.

#include <omp.h>
#include <math.h>

// OpenMP example: root finding

// the function f() is known to be negative
// at a, positive at b, and thus has at least one root in (a,b); if there are multiple roots, only one is found;
// the procedure runs for niter iterations

// strategy: in each iteration, the current interval is split into nth equal parts, and each thread checks its subinterval
// for a sign change of f(); if one is found, this subinterval becomes the new current interval; the current guess for the root is the left endpoint of the current interval
// of course, this approach is useful in parallel only if f() is very expensive to evaluate

// for simplicity, assumes that no endpoint of a subinterval will ever exactly coincide with a root

float root(float (*f)(float),

10 // the matrix is nominally mxn, but its rows will be padded on the right ends,
11 // so as to enable a stride of s down each column; it is assumed that s >= n
12 // allocate space for the padded matrix,
13 // initially empty
14 float *padmalloc(int m, int n, int s) {
15 return (malloc(m*s*sizeof(float))); } 
16
17 // store the value to store in the matrix q,
18 // at row i, column j; m, n and s are as in padmalloc() above
19 void setter(float *q, int m, int n, int s,
20    int i, int j, float tostore) {
21 *(q + i*s+j) = tostore; 
22 }
23
24 // fetch the value in the matrix q,
25 // at row i, column j; m, n and s are as in padmalloc() above
26 float getter(float *q, int m, int n, int s,
27    int i, int j) {
28 return *(q + i*s+j); 
29 }
30
31 float inita, float initb, int niter) {
32 float curra = inita;
33 float currb = initb;
34 #pragma omp parallel 
35 { int nth = omp_get_num_threads();
36 int me = omp_get_thread_num();
37 int iter;
38 for (iter = 0; iter < niter; 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)); } 