```c
/* Subset of these examples adapted from:
   1. http://www.llnl.gov/computing/tutorials/openMP/exercise.html
   2. NAS benchmarks
*/

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

#ifdef _OPENMP
#include <omp.h>
#endif

#define CHUNKSIZE 10
#define N 100

/*****************************************************************************/

void workshare_for()
{
  int nthreads, tid, i, chunk;
  float a[N], b[N], c[N];

  /* initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;

  chunk = CHUNKSIZE;

  #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
    tid = omp_get_thread_num();
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);

    #pragma omp for schedule(dynamic,chunk)
    for (i=0; i<N; i++)
    {
      c[i] = a[i] + b[i];
      printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
    }
  }  /* end of parallel region */
}

/*****************************************************************************/
```

```
void workshare_par()
{
  int   i, tid;
  float a[N], b[N], c[N], d[N];

  for (i=0; i<N; i++)
  {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
    c[i] = d[i] = 0.0;
  }

  #pragma omp parallel shared(a,b,c,d) private(i,tid)
  {
    tid = omp_get_thread_num();

    #pragma omp sections nowait
    {
      #pragma omp section
      {
        printf("Thread %d in section 1\n", tid);
        for (i=0; i<N; i++)
        {
          c[i] = a[i] + b[i];
          printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
        }
      }

      #pragma omp section
      {
        printf("Thread %d in section 2\n", tid);
        for (i=0; i<N; i++)
        {
          d[i] = a[i] * b[i];
          printf("Thread %d: d[%d]= %f\n", tid, i, d[i]);
        }
      }

    }  /* end of sections */

    printf("Thread %d done.\n", tid);

  }  /* end of parallel region */
}

/******************************************************************************/

void reduction()
{
```

```
  int   i;
  float a[N], b[N], sum;

  for (i=0; i < N; i++)
    a[i] = b[i] = i;

  sum = 0.0;

  #pragma omp parallel for reduction(+:sum)
  for (i=0; i < N; i++)
    sum = sum + a[i] * b[i];

  printf("   Sum = %f\n", sum);
}

/*****************************************************************************/

int tid; /* global variable */
#pragma omp threadprivate(tid)

void threadprivate()
{
  #pragma omp parallel
  {
    settid();
    printf("   Thread %d in first region\n", tid);
  }

  #pragma omp parallel
  {
    printf("   Thread %d in second region\n", tid);
  }
}

void settid()
{
  tid = omp_get_thread_num(); /* tid is thread private */
}

/*****************************************************************************/

void matmult()
{
  int     i, j, k, chunk;
  double  a[RA][CA], b[CA][CB], c[RA][CB];

  chunk = CHUNKSIZE;

  #pragma omp parallel shared(a,b,c,chunk) private(i,j,k)
```

```
  {
    #pragma omp sections
    {
      #pragma omp section
      for (i=0; i<RA; i++)
        for (j=0; j<CA; j++)
          a[i][j]= i+j;

      #pragma omp section
      for (i=0; i<CA; i++)
        for (j=0; j<CB; j++)
          b[i][j]= i*j;

      #pragma omp section
      for (i=0; i<RA; i++)
        for (j=0; j<CB; j++)
          c[i][j]= 0;
    } /* end of sections */

    #pragma omp for schedule (static, chunk)
    for (i=0; i<RA; i++)
      for(j=0; j<CB; j++)
        for (k=0; k<CA; k++)
          c[i][j] += a[i][k] * b[k][j];
  } /* end of parallel region */
}

/******************************************************************************/

void heat_diffusion()
{
  int i, n, step, maxsteps;
  double threshold, maxdiff;
  double dx, dt;
  double *uk, *ukp1, *temp;

  /* initializations omitted */

  for (step = 0; (step < maxsteps) && (maxdiff >= threshold); step++)
  {
    /* compute new values */
    #pragma omp parallel for schedule(runtime)
    for (i = 1; i < n-1; i++)
      ukp1[i] = uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);

    /* check for convergence */
    #pragma omp parallel private(i)
    {
      double diff, local_maxdiff = 0.0; /* these are private */
```

```
      #pragma omp single
      maxdiff = 0.0;

      #pragma omp for schedule(runtime)
      for (i = 1; i < n-1; i++)
      {
        diff = fabs(uk[i] - ukp1[i]);
        if (diff > local_maxdiff)
          local_maxdiff = diff;
      }
      #pragma omp critical
      {
        if (local_maxdiff > maxdiff)
          maxdiff = local_maxdiff;
      }
    }

    /* "copy" ukp1 to uk by swapping pointers */
    temp = ukp1; ukp1 = uk; uk = temp;
  }
}

/******************************************************************************/

double f(double x)
{
  return 4.0/(1.0+x*x);
}

void integration()
{
  int i, n;
  double sum, step, x;

  n = 100000;
  step = 1.0/(double)n;

  #pragma omp parallel
  {
    #pragma omp for private(x) reduction(+:sum) schedule(runtime)
    for (i=0; i < n; i++)
    {
      x = (i+0.5)*step;
      sum = sum + f(x);
    }
    #pragma omp master
    {
      printf("Result = %f\n", step * sum);
```

```
    }
  }
}

/*****************************************************************************/

int low, high, tid;
#pragma omp threadprivate(low,high,tid)

#define MAXNZA 30

void sparse()
{
  int i, j, nza;
  int row[N], rowstr[N], col[N][MAXNZA];
  int work, nthreads;

  /* initializations omitted */

  #pragma omp parallel default(shared) private(work)
  {
    tid = omp_get_thread_num();

    #pragma omp master
    nthreads = omp_get_num_threads();

    #pragma omp barrier

    work = (N + nthreads - 1)/nthreads;
    low  = work * tid;
    high = low + work;
    if (high > N)
      high = N;
  }

  /* Adapted from NAS GC benchmark: count the number of triples in each row */
  /* All threads run the same code, but perform assignments to parts of arrays
     that they "own". This speeds up the program, but is not very efficient.
  */
  #pragma omp parallel default(shared) private(i,j,nza)
  {
    /* owner computes */
    for (j = low; j < high; j++)
      rowstr[j] = 0;

    for (i = 0; i < N; i++)
    {
      for (nza = 0; nza < row[i]; nza++);
      {
```

```
        j = col[i][nza];
        /* owner computes */
        if (j >= low && j < high)
          rowstr[j] = rowstr[j] + row[i];
      }
    }
  }
}
```