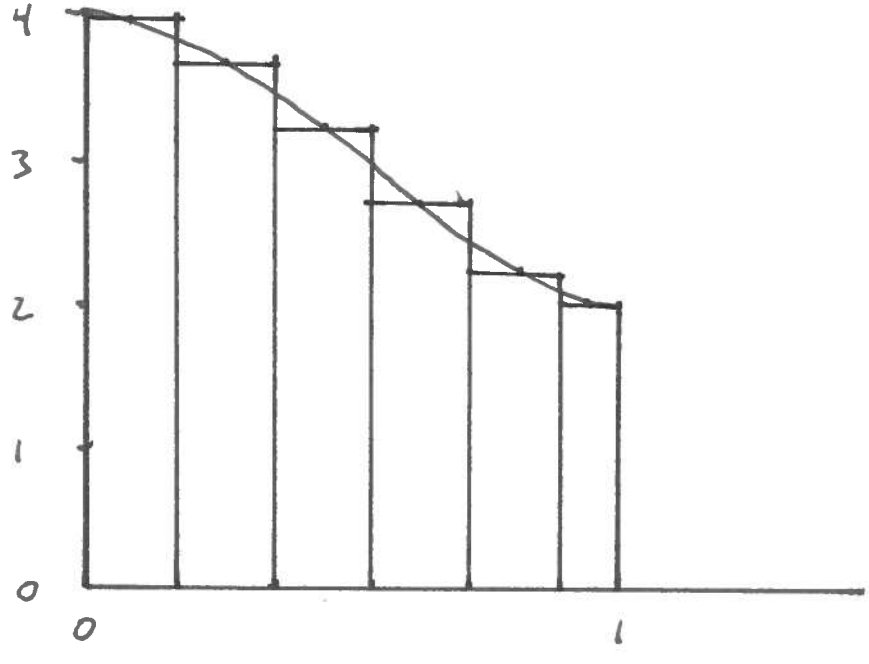
POSIX Threads

CS520

Dept. of Computer Science
Univ. of New Hampshire

Computing π

numeric integration: $\int_0^1 \frac{4}{1+x^2}$



pi0.c

all programs available on ag.fe in ucs520/public/pi-pos

```
/* Pi computation in C (90.11.09) */

#include <stdio.h>

#define INTERVALS 50000000

main (argc, argv)
   int argc; char *argv[];
{
   int i;
   double sum;          /* Sum of areas */
   double width;        /* Width of interval */
   double x;            /* Midpoint of rectangle on x axis */

   start_timer();

   width = 1.0 / INTERVALS;

   sum = 0.0;
   x = width * .5;
   for (i = 0; i < INTERVALS; i++) {
      sum += 4.0/(1.0+x*x);
      x += width;
   }
   sum *= width;

   stop_timer();

   printf ("Estimation of pi is %14.12f\n", sum);
}
```

```
// pi4.c (Uses Posix threads on Linux, Sept 12)
// child threads put partial sum in an array
// thread_join used to have main thread wait
// main thread sums values in array to get final sum

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <pthread.h>

// how many intervals to divide [0,1] into
#define INTERVALS 50000000

// width of an interval
#define WIDTH (1.0/INTERVALS)

// the maximum number of child threads that can be used
#define MAX_T 24

// array for storing thread IDs
pthread_t pt[MAX_T];

// child threads drop values into the array
double partialSums[MAX_T];

// number of intervals per thread
int chunk;

// processor IDs less than split have one extra interval
int split;

// prints usage message if command line arguments are not what is expected
// program expects a single argument: the number of child threads to create
void usage(char *);

// prints an error message and terminates the program
void error(char *);

// this is the work executed by each child thread
void * work(void *);

// timer functions
void start_timer(void);
void stop_timer(void);

int main (int argc, char *argv[])
{
   int i;
   int n; // number of child threads to create */

   if (argc !=2) usage("pi numofthreads\n");

   n = atoi(argv[1]);

   if (n <= 0) usage("number of threads must be > 0");

   if (n > MAX_T) usage("too many threads requested");

   printf("%d (child) threads used\n", n);

   start_timer();

   chunk = INTERVALS / n;
   split = INTERVALS % n;
   if (split == 0)
```

```
   {
      split = n;
      chunk -= 1;
   }

   // create n threads
   for (i=0; i< n; i++)
   {
      /* create threads; DANGER: thread logical id (int) passed as "void *" */
      if (pthread_create(&pt[i], NULL, work, (void *) i) != 0)
         error("error in thread create");
   }

   double sum = 0.0;

   // wait for each thread to finish
   for (i=0; i < n; i++)
   {
     if (pthread_join(pt[i], NULL))
     {
       error("error in thread join");
     }
     sum += partialSums[i];
   }
   sum *= 1.0/INTERVALS;

   stop_timer();

   printf ("Estimation of pi is %14.12f\n", sum);

   return 0;
}

void * work(void * in)
{
   int i;
   int low;                  // first interval to be processed
   int high;                 // first interval *not* to be processed
   double localSum = 0.0;    // sum for intervals being processed
   double x;                 // mid-point of an interval
   long id = (long) in;      // logical thread id (0..n-1)

   if (id < split)
   {
      low = (id * (chunk + 1));
      high = low + (chunk + 1);
   }
   else
   {
      low = (split * (chunk + 1)) + ((id - split) * chunk);
      high = low + chunk;
   }

   x = (low+0.5)*WIDTH;
   for (i = low; i < high; i++)
   {
      localSum += (4.0/(1.0+x*x));
      x += WIDTH;
   }

   partialSums[id] = localSum;
}

void error(char *str)
{
    perror(str);
```

man pgs for all these pthread routines

Function pointer the "work" function

```
    exit(-1);
}

void usage(char *str)
{
    fprintf(stderr, "usage: %s\n",str);
    exit(-1);
}
```

Performance

p.0	serial	.330 sec
	1 thread	.330
	2 threads	.170
[redacted]		
p.3	4 threads	.090
p.4		
p.5	8 threads	.100
	16 threads	.100

← must be 4 CPUs on the machine I am running on

```
// pi5.c (Uses Posix threads on Linux, Sept 12)
// child threads update global sum using mutex to protect the update
// thread_join used to have main thread wait
// main thread then prints the global sum

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <pthread.h>

// how many intervals to divide [0,1] into
#define INTERVALS 50000000

// width of an interval
#define WIDTH (1.0/INTERVALS)

// the maximum number of child threads that can be used
#define MAX_T 24

// global sum, update by threads using mutex to protect
double sum;

// array for storing thread IDs
pthread_t pt[MAX_T];

// mutex to protect update to global sum
pthread_mutex_t mu;

// number of intervals per thread
int chunk;

// processor IDs less than split have one extra interval
int split;

// prints usage message if command line arguments are not what is expected
// program expects a single argument: the number of child threads to create
void usage(char *);

// prints an error message and terminates the program
void error(char *);

// this is the work executed by each child thread
void * work(void *);

// timer functions
void start_timer(void);
void stop_timer(void);

int main (int argc, char *argv[])
{
   int i;
   int n; // number of child threads to create */

   if (argc !=2) usage("pi numofthreads\n");

   n = atoi(argv[1]);

   if (n <= 0) usage("number of threads must be > 0");

   if (n > MAX_T) usage("too many threads requested");

   printf("%d (child) threads used\n", n);

   start_timer();
```

```
   chunk = INTERVALS / n;
   split = INTERVALS % n;
   if (split == 0)
   {
      split = n;
      chunk -= 1;
   }

   // initialize global sum
   sum = 0.0;

   // initialize mutex to protect global sum
   if (pthread_mutex_init(&mu, NULL) != 0)
      error("can't init mutex");

   // create n threads
   for (i=0; i < n; i++)
   {
      /* create threads; DANGER: thread logical id (int) passed as "void *" */
      if (pthread_create(&pt[i], NULL, work, (void *) i) != 0)
         error("error in thread create");
   }

   // wait for each thread to finish
   for (i=0; i < n; i++)
   {
     if (pthread_join(pt[i], NULL))
     {
       error("error in thread join");
     }
   }
   sum *= 1.0/INTERVALS;

   stop_timer();

   printf ("Estimation of pi is %14.12f\n", sum);

   return 0;
}

void * work(void * in)
{
   int i;
   int low;                  // first interval to be processed
   int high;                 // first interval *not* to be processed
   double localSum = 0.0;    // sum for intervals being processed
   double x;                 // mid-point of an interval
   long id = (long) in;      // logical thread id (0..n-1)

   if (id < split)
   {
      low = (id * (chunk + 1));
      high = low + (chunk + 1);
   }
   else
   {
      low = (split * (chunk + 1)) + ((id - split) * chunk);
      high = low + chunk;
   }

   x = (low+0.5)*WIDTH;
   for (i = low; i < high; i++)
   {
      localSum += (4.0/(1.0+x*x));
      x += WIDTH;
   }
```

```
    if (pthread_mutex_lock(&mu) != 0)
        error("error in mutex_lock in child");

    sum += localSum;

    if (pthread_mutex_unlock(&mu) != 0)
        error("error in mutex_unlock in child");
}

void error(char *str)
{
     perror(str);
     exit(-1);
}

void usage(char *str)
{
    fprintf(stderr, "usage: %s\n",str);
    exit(-1);
}
```

```
// pi3.c (Uses Posix threads on Linux, Oct 10)
//
// This version uses a mutex and condition variable to coordinate
// update to a shared variable and to communicate to the parent
// thread when all child threads are done. Also the parent does
// part of the work. It acts like child 0.

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <pthread.h>

#define INTERVALS 50000000

#define MAX_T 24

pthread_t pt[MAX_T];

pthread_mutex_t mu;              /* Mutex for "sum" and "bar_cnt" */

pthread_cond_t cv;               /* Condition variable to hold parent
                                       until all children done */

int cnt;                         /* Counter for exiting children */

double sum;                      /* Global sum of areas */

double width = 1.0/INTERVALS;    /* Width of interval */

int chunk;                       /* Number of intervals per thread */

int split;                       /* Processor ids less than split
                                       have one extra interval     */

int n;                           /* Number of threads to create */

void usage(char *);

void error(char *);

void * work(void *);

int main (int argc, char *argv[])
{
   long i;

   if (argc != 2) usage("pi numofthreads\n");

   n = atoi(argv[1]);

   if (n <= 0) usage("number of threads must be > 0");

   printf("%d threads used\n", n);

   start_timer();

   chunk = INTERVALS / n;
   split = INTERVALS % n;
   if (split == 0)
   {
      split = n;
      chunk -= 1;
   }

   if (pthread_mutex_init(&mu, NULL) != 0)
```

```
      error("can't init mutex");

   if (pthread_cond_init(&cv, NULL) != 0)
      error("can't init condition variable");

   cnt = 0;    // number of children that have exited

   sum = 0.0;

   for (i=1; i < n; i++)
   {
      // create threads; DANGER: thread logical id (int) passed as void*
      if (pthread_create(&pt[i], NULL, work, (void *) i) != 0)
         error("error in thread create");
   }

   work(0);

   if (++cnt != n)
   {
      // wait for all children to finish
      if (pthread_cond_wait(&cv, &mu) != 0)
         error("error in cond_wait by parent");
   }

   sum *= 1.0/INTERVALS;

   stop_timer();

   printf ("Estimation of pi is %14.12f\n", sum);

   return 0;
}

void * work(void * in)
{
   int i;
   int low;                        /* First interval to be processed */
   int high;                       /* First interval *not* to be processed */
   double local_sum = 0.0;         /* Sum for intervals being processed */
   double x;                       /* Mid-point of an interval    */
   long id = (long) in;            /* Logical thread id (0..n-1) */

   if (id < split)
   {
      low = (id * (chunk + 1));
      high = low + (chunk + 1);
   }
   else
   {
      low = (split * (chunk + 1)) + ((id - split) * chunk);
      high = low + chunk;
   }

   x = (low+0.5)*width;
   for (i = low; i < high; i++)
   {
      local_sum += (4.0/(1.0+x*x));
      x += width;
   }

   if (pthread_mutex_lock(&mu) != 0)
      error("error in mutex_lock");

   sum += local_sum;
```

Main thread is doing some of the work!

```
    if (id == 0)  // parent continues to hold lock!
        return;

    if (++cnt == n)  // all threads are now done
    {
        if (pthread_cond_signal(&cv) != 0)  // wakeup parent
              error("error in cond_signal");
    }

    if (pthread_mutex_unlock(&mu) != 0)
        error("error in mutex_unlock");

}

void error(char *str)
{
     perror(str);
     exit(-1);
}

void usage(char *str)
{
    fprintf(stderr, "usage: %s\n",str);
    exit(-1);
}
```

main thread