CSE 4351 Notes 2: Data Parallel Programing in LinuxThreads (pthreads)

LinuxThreads/Threads

Threads are main unit of concurrent execution

Lightweight processes are available in some implementations

LinuxThreads are implemented as heavyweight processes

LinuxThreads/Compilation

```
cc -D_REENTRANT -o name name.c -lm -lpthread

cc -D_REENTRANT -o name name.c barrier.c -lm -lpthread /* if barriers are used */

#include <pthread.h>      /* pthread header file */
```

LinuxThreads Primitives/Creating Threads

`pthread_t` type for storing thread related data

```
int pthread_create(pthread_t * thread, pthread_attr_t *attr,
      void * (start_routine)(void *), void *arg);
```

Returns 0 if successful, passes 1) thread related data, 2) creation attributes (usually NULL),

3) pointer to function, 4) pointer to argument structure (number of processes, rank of this process, data)

Parallel Primitives/Parallel I/O

AVOID CONCURRENT I/O ON A FILE DESCRIPTOR

Convention: When reasonable, let process with local id=0 (the parent) do all I/O

Debugging/Tracing: Use a lock (covered later) to prevent concurrent I/O

On occasion fflush(stdout) is needed.

Parallel Primitives/Barriers

`#include "barrier.h"` header file for Butenhof's barrier code (must be in same directory as your source code)

`barrier_t barrier;` Declares a barrier named "barrier"

`barrier_init(&barrier,i);` Initializes barrier indicating that i processes must check in

`barrier_wait(&barrier);` Causes thread to block until the others have checked-in at same barrier

Parallel Primitives/Mutexes

`pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;` Declares a mutex named "lock"

LOCKS ARE LOCKS, DATA IS DATA

THE PROGRAMMER(S) DETERMINE THE RELEVANCE OF LOCKS

`pthread_mutex_init(ptr, NULL);`	Initializes lock.
`pthread_mutex_lock(&lock);`	Wait & set lock. `pthread_mutex_trylock(&lock)` will not wait.
`pthread_mutex_unlock(&lock);`	Clear lock.

Example/Summing

```
$cat serial.c
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

int i,k;
double sum;

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

main()
{
  float startCPU,startTime;

  printf("enter k (in millions):");
  scanf("%d",&k);
  k *= 1000000;
  printf("k is %d\n",k);
  startCPU=CPUtime();
  startTime=elapsedTime();
  sum=0;
  for (i=0;i<=k;i++)
    sum+=i;
  printf("sum is %f\n",sum);
  printf("should be %lf\n",(double)(k)*(double)(k+1)/2.0);
  printf("CPU %f\n",CPUtime()-startCPU);
  printf("Elapsed %f\n",elapsedTime()-startTime);
}
```

Parallel addition of a list of numbers WITHOUT SYNCHRONIZATION.

```
$ cat nolockPT.c
/* Simple pthreads example to incorrectly add numbers */
#include <pthread.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "barrier.h"

barrier_t barrier;

int k;

double sum;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;
```

```
float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void *sumEmUp(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i;
float startCPU,startTime;

barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
for (i=threadWork->threadId;i<=k;i+=numThreads)
{
  sum+=i;
}
printf("CPU used by thread %d is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("Elapsed time for thread %d is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
barrier_wait(&barrier);
}

main()
{
thread_data_t *threadWork;
int i,status;
pthread_t thread;

printf("enter number of threads:");
scanf("%d",&numThreads);
barrier_init (&barrier, numThreads);
printf("enter k (in millions):");
scanf("%d",&k);
k *= 1000000;
printf("k is %d\n",k);
sum=0.0;

for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  status = pthread_create (
              &thread, NULL, sumEmUp, threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
sumEmUp((void*) threadWork);

printf("sum is %f\n",sum);
printf("should be %f\n",(double)(k)*(double)(k+1)/2.0);
}
```

Parallel addition of a list of numbers WITH TOO MUCH LOCKING.

```
$ cat mutexPT.c
/* Simple pthreads example to correctly add numbers but
```

```
   with excessive use of a mutex */
#include <pthread.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "barrier.h"

barrier_t barrier;

int k;

double sum;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void *sumEmUp(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i;
float startCPU,startTime;

barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
for (i=threadWork->threadId;i<=k;i+=numThreads)
{
  pthread_mutex_lock(&mutex);
  sum+=i;
  pthread_mutex_unlock(&mutex);
}
printf("CPU used by thread %d is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("Elapsed time for thread %d is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
barrier_wait(&barrier);
}

main()
{
thread_data_t *threadWork;
int i,status;
pthread_t thread;

printf("Enter number of threads\n");
scanf("%d",&numThreads);
if (barrier_init (&barrier, numThreads))
  printf("barrier_init problem\n");
printf("enter k (in millions):");
scanf("%d",&k);
k *= 1000000;
printf("k is %d\n",k);
sum=0.0;

for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
```

```
  threadWork->threadId=i;
  status = pthread_create (
               &thread, NULL, sumEmUp,  threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
sumEmUp((void*) threadWork);

printf("sum is %f\n",sum);
printf("should be %f\n",(double)(k)*(double)(k+1)/2.0);
}
```

Parallel addition of a list of numbers WITH LOCAL ACCUMULATION.

```
$ cat smartMutexPT.c
/* Simple pthreads example to correctly add numbers but
   with restricted use of a mutex */
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <pthread.h>
#include "barrier.h"

barrier_t barrier1,barrier2;

int k;

double sum;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void *sumEmUp(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i;
double localSum=0.0;
float startCPU,startTime;

barrier_wait(&barrier1);
startCPU=CPUtime();
startTime=elapsedTime();
for (i=threadWork->threadId;i<=k;i+=numThreads)
{
  localSum+=i;
}
pthread_mutex_lock(&mutex);
sum+=localSum;
pthread_mutex_unlock(&mutex);
printf("CPU used by thread %d is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
```

```
printf("Elapsed time for thread %d is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
barrier_wait(&barrier2);
}

main()
{
thread_data_t *threadWork;
int i,status;
pthread_t thread;

printf("enter number of threads:");
scanf("%d",&numThreads);
barrier_init(&barrier1,numThreads);
barrier_init(&barrier2,numThreads+1);
printf("enter k (in millions):");
scanf("%d",&k);
k *= 1000000;
printf("k is %d\n",k);
sum=0.0;

for (i=0;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  status = pthread_create (
                &thread, NULL, sumEmUp, threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

barrier_wait(&barrier2);

printf("sum is %f\n",sum);
printf("should be %f\n",(double)(k)*(double)(k+1)/2.0);
}
```

Comparison of Various Versions:

```
[bob@ketchup NOTES02]$ cat runScript
echo 'serial<100'
serial<100
echo 'nolockPT<2.100'
nolockPT<2.100
echo 'mutexPT<2.1'
mutexPT<2.1
echo 'smartMutexPT<2.100'
smartMutexPT<2.100
[bob@ketchup NOTES02]$ cat 100
100
[bob@ketchup NOTES02]$ cat 2.100
2
100
[bob@ketchup NOTES02]$ cat 2.1
2
1
[bob@ketchup NOTES02]$ source runScript
serial<100
enter k (in millions):k is 100000000
sum is 5000000050000000.000000
should be 5000000050000000.000000
CPU 2.080000
Elapsed 2.000000
nolockPT<2.100
enter number of threads:enter k (in millions):k is 100000000
CPU used by thread 0 is 8.380000
Elapsed time for thread 0 is 8.500000
CPU used by thread 1 is 8.380000
Elapsed time for thread 1 is 8.500000
sum is 3981834892681993.000000
```

```
should be 5000000050000000.000000
mutexPT<2.1
Enter number of threads
enter k (in millions):k is 1000000
CPU used by thread 1 is 13.540000
Elapsed time for thread 1 is 24.500000
CPU used by thread 0 is 12.920000
Elapsed time for thread 0 is 24.750000
sum is 500000500000.000000
should be 500000500000.000000
smartMutexPT<2.100
enter number of threads:enter k (in millions):k is 100000000
CPU used by thread 1 is 1.070000
Elapsed time for thread 1 is 1.079102
CPU used by thread 0 is 1.080000
Elapsed time for thread 0 is 1.079102
sum is 5000000050000000.000000
should be 5000000050000000.000000
```

Concurrent Loops

All previous examples are based on parallelizing a single loop using interleaving ("horizontal spreading"). This is the simplest form of static scheduling (or prescheduling). Given a *simple* for-loop without dependencies between iterations:

```
for (i=0; i<limit; i++)
{ . . . }
```

it is converted to:

```
for (i=threadWork->threadId; i<limit; i+=numThreads)
{ . . . }
```

Contiguous form ("vertical spreading") is another form of static scheduling that may be more tedious to code:

```
lower = threadWork->threadId * limit / numThreads;
upper = (threadWork->threadId + 1) * limit / numThreads - 1;
for (i=lower; i<=upper; i++)
{ . . . }
```

There are four typical assumptions for using these two methods: 1) each iteration will take about the same amount of time, 2) the loop overhead is insignificant compared to the time to execute the loop body, 3) the loop control variable (e.g. i in the example) is modified in a *simple* way (increment/addition in the example), and 4) all participating processes reach the loop simultaneously. When any of these assumptions are broken, then dynamic scheduling (or self-scheduling or worklists) may be useful:

```
if (threadWork->threadId==0)
      i = sharedI = 0;
barrier_wait(&barrier);
if (threadWork->threadId!=0)
{
      pthread_mutex_lock(&iLock);
      i = ++sharedI;
      pthread_mutex_unlock(&iLock);
}
while (i<limit)
{
      ...
      pthread_mutex_lock(&iLock);
      i = ++sharedI;
      pthread_mutex_unlock(&iLock);
}
```

These techniques must be specialized to given situations and may include other forms of loops. In the worst case, a loop must be completely rewritten based on an understanding of the code. A particularly nasty situation involves multiple loop-variables as in the following simple example:

```
k=n-1;
for (i=0;i<n;i++)
{
      b[k]=a[i];
      k--;
}
```

The interleaved version of this loop is:

```
k=n-1-threadWork->threadID;
for (i=threadWork->threadID;i<n;i+=numThreads)
{
      b[k]=a[i];
      k-=numThreads;
}
```

The contiguous version of this loop is:

```
ilower=threadWork->threadID*n/numThreads;
iupper=(threadWork->threadID+1)*n/numThreads-1;
k=n-1-ilower;
for (i=ilower;i<=iupper;i++)
{
      b[k]=a[i];
      k--;
}
```

The dynamic version is:

```
if (threadWork->threadID==0)
{
      i = sharedI = 0;
      k = sharedK = n-1;
}
barrier_wait(&barrier);
if (threadWork->threadID!=0)
{
      pthread_mutex_lock(&iLock);
      i = ++sharedI;
      k = --sharedK;
      pthread_mutex_unlock(&iLock);
}
while (i<n)
{
      b[k]=a[i];
      pthread_mutex_lock(&iLock);
      i = ++sharedI;
      k = --sharedK;
      pthread_mutex_unlock(&iLock);
}
```

Example/Find Minimum

```
$ cat min.c
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

int minimum(a,N)
int *a;
int N;
{
  int i,min;
  float startCPU,startTime;
```

```
  startCPU=CPUtime();
  startTime=elapsedTime();
  min=a[0];
  for (i=1;i<N;i++)
    if (a[i]<min)
      min=a[i];
  printf("CPU used is %f\n",CPUtime()-startCPU);
  printf("Elapsed time is %f\n",elapsedTime()-startTime);
  return min;
}

main()
{
  int *arr;
  int i,arrSize,min;
  unsigned seed;
  printf("enter number of keys\n");
  scanf("%d",&arrSize);
  arr=(int*) malloc(sizeof(int)*arrSize);
  if (!arr)
  {
    printf("malloc failed\n");
    abort();
  }
  printf("enter seed\n");
  scanf("%u",&seed);
  srandom(seed);
  for (i=0;i<arrSize;i++)
    arr[i]=random();
  min=minimum(arr,arrSize);
  printf("The minimum value is %d\n",min);
}

$ cat minPT.c
/* Find minimum of set of numbers - uses atomicity of memory accesses */
#include <pthread.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "barrier.h"

barrier_t barrier;

int min;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

int numThreads;

typedef struct thread_data {
  int threadId;
  int *a;
  int N;
} thread_data_t;

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void *minimum(void* arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int *a;
int N;
int i;
int lower,upper;
```

```
float startCPU,startTime;

a=threadWork->a;
N=threadWork->N;
barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
lower=threadWork->threadId * N / numThreads;
upper=(threadWork->threadId + 1) * N / numThreads - 1;
for (i=lower;i<=upper;i++)
  if (a[i]<min)
  {
    pthread_mutex_lock(&mutex);
    if (a[i]<min)
      min=a[i];
    pthread_mutex_unlock(&mutex);
  }
printf("CPU used by thread %d is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("Elapsed time for thread %d is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
barrier_wait(&barrier);
}

main()
{
thread_data_t *threadWork;
int *arr;
int i,arrSize,status;
unsigned seed;
pthread_t thread;

printf("enter number of threads:");
scanf("%d",&numThreads);
if (barrier_init (&barrier, numThreads))
  printf("barrier_init problem\n");;
printf("enter number of keys\n");
scanf("%d",&arrSize);
arr=(int*) malloc(sizeof(int)*arrSize);
if (!arr)
{
  printf("malloc failed\n");
  abort();
}
printf("enter seed\n");
scanf("%u",&seed);
srandom(seed);
for (i=0;i<arrSize;i++)
  arr[i]=random();
min=arr[0];

for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  threadWork->a=arr;
  threadWork->N=arrSize;
  status = pthread_create (
                &thread, NULL, minimum, (void*) threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
threadWork->a=arr;
threadWork->N=arrSize;
minimum((void*) threadWork);

printf("The minimum value is %d\n",min);
}
```

```
[bob@ketchup NOTES02]$ min
enter number of keys
50000000
enter seed
567
CPU used is 1.830000
Elapsed time is 1.750000
The minimum value is 13
[bob@ketchup NOTES02]$ minPT
enter number of threads:2
enter number of keys
50000000
enter seed
567
CPU used by thread 0 is 1.059999
Elapsed time for thread 0 is 1.000000
CPU used by thread 1 is 1.060000
Elapsed time for thread 1 is 1.000000
The minimum value is 13
```

Example/Shellsort

```
$ cat shellserial.c
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <sys/resource.h>

void shellsort(a,N)
int *a;
int N;
{
  int group;
  int i,j,h;
  int v;
  for (h=1;h<=N/9; h=3*h+1) ;
  for (;h>0; h /= 3)
    for (group=0; group<h; group++)
       for (i=group+h; i<N; i+=h)
       {
         v=a[i];
         j=i;
         while (j>=h && a[j-h]>v)
         {
           a[j]=a[j-h];
           j -= h;
         }
         a[j]=v;
       }
}

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

main()
{
  int *arr;
  int i,arrSize;
  unsigned seed;
  float startCPU,startTime;

  printf("enter number of keys\n");
```

```
  scanf("%d",&arrSize);
  arr=(int*) malloc(sizeof(long)*arrSize);
  if (!arr)
  {
    printf("malloc failed\n");
    abort();
  }
  printf("enter seed\n");
  scanf("%lu",&seed);
  srandom(seed);
  for (i=0;i<arrSize;i++)
    arr[i]=random();
  startCPU=CPUtime();
  startTime=elapsedTime();
  shellsort(arr,arrSize);
  printf("CPU used is %f\n",CPUtime()-startCPU);
  printf("Elapsed time is %f\n",elapsedTime()-startTime);
}

$ cat shellPT.c
/* Shellsort */
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "barrier.h"

barrier_t barrier;

int *arr;
int arrSize;

int numThreads;

typedef struct thread_data {
  int threadId;
  int *a;
  int N;
} thread_data_t;

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void *shellsort(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int *a;
int N;

int group;
int i,j,h;
int v;
float startCPU,startTime;

a=threadWork->a;
N=threadWork->N;
barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
for (h=1;h<=N/9; h=3*h+1) ;
for (;h>0; h /= 3)
{
  for (group=threadWork->threadId; group<h; group+=numThreads)
```

```
    for (i=group+h; i<N; i+=h)
    {
      v=a[i];
      j=i;
      while (j>=h && a[j-h]>v)
      {
        a[j]=a[j-h];
        j -= h;
      }
      a[j]=v;
    }
  barrier_wait(&barrier);
}
printf("CPU used by thread %d is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("Elapsed time for thread %d is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
barrier_wait(&barrier);
}

main()
{
thread_data_t *threadWork;
pthread_t thread;
int i,status;
unsigned seed;

printf("enter number of threads:");
scanf("%d",&numThreads);
if (barrier_init (&barrier, numThreads))
  printf("barrier_init problem\n");;
printf("enter number of keys\n");
scanf("%d",&arrSize);
arr=(int*) malloc(sizeof(int)*arrSize);
if (!arr)
{
  printf("malloc failed\n");
  abort();
}
printf("enter seed\n");
scanf("%u",&seed);
srandom(seed);
for (i=0;i<arrSize;i++)
  arr[i]=random();

for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  threadWork->a=arr;
  threadWork->N=arrSize;
  status = pthread_create (
              &thread, NULL, shellsort, (void*) threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
threadWork->a=arr;
threadWork->N=arrSize;
shellsort((void*) threadWork);
printf("Verifying sort . . .\n");
for (i=1;i<arrSize;i++)
  if (arr[i-1]>arr[i])
    break;
if (i<arrSize)
  printf("sort failed\n");
else
  printf("sort succeeded\n");
}
```

```
[bob@ketchup NOTES02]$ shellserial
enter number of keys
2000000
enter seed
123
CPU used is 16.650001
Elapsed time is 16.750000
[bob@ketchup NOTES02]$ shellPT
enter number of threads:2
enter number of keys
2000000
enter seed
123
CPU used by thread 0 is 9.620000
CPU used by thread 1 is 8.720000
Elapsed time for thread 1 is 9.750000
Elapsed time for thread 0 is 9.750000
Verifying sort . . .
sort succeeded
```

Example/Matrix Multiplication

```
$ cat mm.c
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

double a[500][500],b[500][500],c[500][500];

void wholeThing()
{
int i,j,k;
double sum;

for (i=0;i<500;i++)
  for (j=0;j<500;j++)
  {
    a[i][j]=1.0/(i+j+2);
    b[i][j]=i+j;
  }

for (i=0;i<500;i++)
  for (j=0;j<500;j++)
  {
    sum=0.0;
    for (k=0;k<500;k++)
      sum+=a[i][k]*b[k][j];
    c[i][j]=sum;
  }
}

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

main()
{
float startCPU,startTime;

startCPU=CPUtime();
startTime=elapsedTime();
wholeThing();
printf("CPU used is %f\n",CPUtime()-startCPU);
printf("Elapsed time is %f\n",elapsedTime()-startTime);
```

```
}

$ cat mmPT.c
#include <stdio.h>
 #include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "barrier.h"

barrier_t barrier;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;

float elapsedCPU()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

double a[500][500],b[500][500],c[500][500];

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void *wholeThing(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i,j,k;
double sum;
int id;
float startCPU,startTime;

id=threadWork->threadId;
barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
for (i=id;i<500;i+=numThreads)
  for (j=0;j<500;j++)
  {
    a[i][j]=1.0/(i+j+2);
    b[i][j]=i+j;
  }
barrier_wait(&barrier);
for (i=id;i<500;i+=numThreads)
  for (j=0;j<500;j++)
  {
    sum=0.0;
    for (k=0;k<500;k++)
      sum+=a[i][k]*b[k][j];
    c[i][j]=sum;
  }
printf("CPU used by thread %d is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("Elapsed time for thread %d is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
barrier_wait(&barrier);
}
```

```
main()
{
int i,j,k;
thread_data_t *threadWork;
pthread_t thread;
int status;
double sum;

printf("enter number of threads:");
scanf("%d",&numThreads);
if (barrier_init (&barrier, numThreads))
  printf("barrier_init problem\n");;
for (i=1;i<numThreads;i++)
{
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=i;
  status = pthread_create (
              &thread, NULL, wholeThing, (void*) threadWork);
  if (status != 0)
  {
    printf("failed to create thread\n");
    exit(0);
  }
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;
wholeThing((void*) threadWork);
printf("verifying . . .\n");
for (i=0;i<500;i++)
  for (j=0;j<500;j++)
  {
    sum=0.0;
    for (k=0;k<500;k++)
      sum+=a[i][k]*b[k][j];
    if (c[i][j]!=sum)
      printf("Problem with c[%d][%d]=%f != %f\n",i,j,c[i][j],sum);
  }
}
```

```
[bob@ketchup NOTES02]$ mm
CPU used is 27.360001
Elapsed time is 27.250000
[bob@ketchup NOTES02]$ mmPT
enter number of threads:2
CPU used by thread 1 is 10.590000
Elapsed time for thread 1 is 10.750000
CPU used by thread 0 is 13.910000
Elapsed time for thread 0 is 14.000000
verifying . . .
```

Example/Quicksort:

```
[weems@va NOTES02]$ cat qsort.c
// Ordinary recursive quicksort
// See CLR for details
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}
```

```
int *arr;
int arrSize;

int partition(arr,p,r)
int *arr,p,r;
{
int x,i,j,temp;

/*
printf("Input\n");
for (i=p;i<=r;i++)
  printf("%d %d\n",i,arr[i]);
*/
x=arr[p];
i=p-1;
j=r+1;
while (1)
{
  while (arr[--j]>x);
  while (arr[++i]<x);
  if (i<j)
  {
    temp=arr[i];
    arr[i]=arr[j];
    arr[j]=temp;
  }
  else
  {
/*
    printf("Left output\n");
    for (i=p;i<=j;i++)
      printf("%d %d\n",i,arr[i]);
    printf("Right output\n");
    for (i=j+1;i<=r;i++)
      printf("%d %d\n",i,arr[i]);
*/
    return j;
  }
}
}

quickSort(int *a,int p,int r)
{
int q;

if (p<r)
{
  q=partition(a,p,r);
  quickSort(a,p,q);
  quickSort(a,q+1,r);
}
}

main()
{
int i;
unsigned seed;
float startCPU,startTime;

printf("enter number of keys\n");
scanf("%d",&arrSize);
arr=(int*) malloc(sizeof(int)*arrSize);
if (!arr)
{
  printf("malloc failed\n");
  abort();
}
printf("enter seed\n");
scanf("%u",&seed);
srandom(seed);
for (i=0;i<arrSize;i++)
  arr[i]=random();
startCPU=CPUtime();
```

```
startTime=elapsedTime();
quickSort(arr,0,arrSize-1);
printf("CPU used is %f\n",CPUtime()-startCPU);
printf("Elapsed time is %f\n",elapsedTime()-startTime);
}
```

Concurrent version is designed such that thread 0 starts the quicksort and will pass small partitions to thread 1 through a single-slot buffer. Thread 1 completely quicksorts the partitions that are pass to it.

Shared variables:

`zeroActive` - initially set to 1. Thread 0 sets this to 0 when it is no longer participating in the sort.

`partitionAvailable` - initially set to 0. When `partitionAvailable` is 0, thread 0 may pass a subproblem to thread 1 by setting the values of `leftEnd` and `rightEnd` to indicate the subarray bounds and then set `partitionAvailable` to 1 to complete the hand-off. Thread 1 will reset `partitionAvailable` to 0 when the subproblem has been completely sorted.

Thread 0:

`quickSort0()` initially calls `quickSort2()`
`quickSort2()` will pass a ‘‘small’’ subproblem to thread 1 when thread 1 is waiting

Thread 1:

`quickSort1()` spins waiting for a subproblem to work on, then uses sequential `quickSort()`

```
[weems@va NOTES02]$ cat qsortPT.c
// Recursive quicksort using pthreads
// See CLR for details
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <pthread.h>
#include "barrier.h"

barrier_t barrier;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

int *arr;
int arrSize;
int zeroActive=1,partitionAvailable=0,leftEnd,rightEnd;
int maxThreshold;

int partition(arr,p,r)
int *arr,p,r;
{
int x,i,j,temp;

/*
printf("Input\n");
for (i=p;i<=r;i++)
  printf("%d %d\n",i,arr[i]);
```

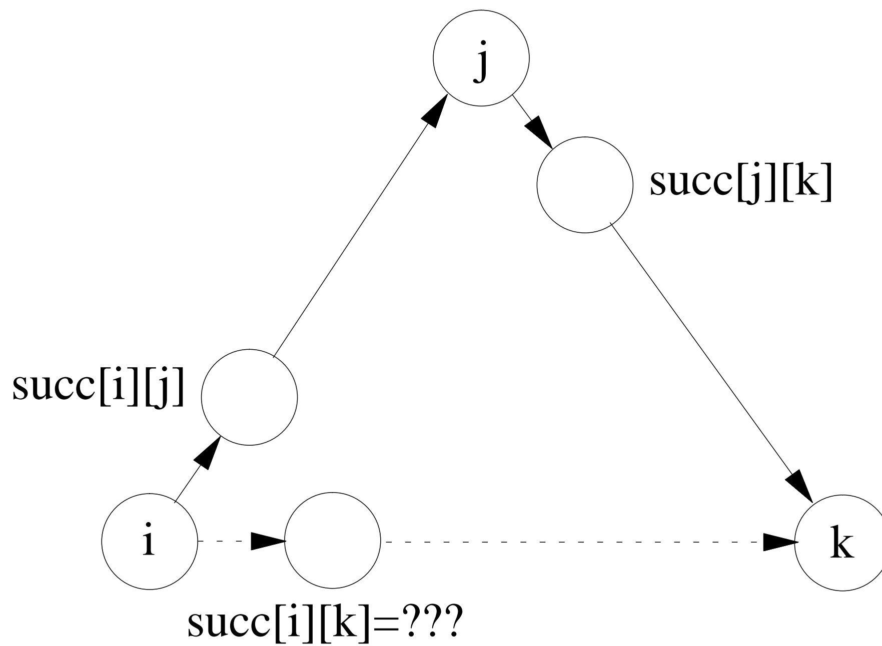
```
*/
x=arr[p];
i=p-1;
j=r+1;
while (1)
{
  while (arr[--j]>x);
  while (arr[++i]<x);
  if (i<j)
  {
    temp=arr[i];
    arr[i]=arr[j];
    arr[j]=temp;
  }
  else
  {
/*
    printf("Left output\n");
    for (i=p;i<=j;i++)
      printf("%d %d\n",i,arr[i]);
    printf("Right output\n");
    for (i=j+1;i<=r;i++)
      printf("%d %d\n",i,arr[i]);
*/
    return j;
  }
}
}

quickSort(int *a,int p,int r)
{
int q;

if (p<r)
{
  q=partition(a,p,r);
  quickSort(a,p,q);
  quickSort(a,q+1,r);
}
}

quickSort2(int *a,int p,int r)
{
int q,k;

if (p<r)
{
  q=partition(a,p,r);
  k=r-p+1;
  if (partitionAvailable || k>maxThreshold)
    quickSort2(a,p,q);
  else
  {
    leftEnd=p;
    rightEnd=q;
    partitionAvailable=1;
  }
  k=r-q;
  if (partitionAvailable || k>maxThreshold)
    quickSort2(a,q+1,r);
  else
  {
    leftEnd=q+1;
    rightEnd=r;
    partitionAvailable=1;
  }
}
}

void *quickSort0(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
float startCPU,startTime;
```

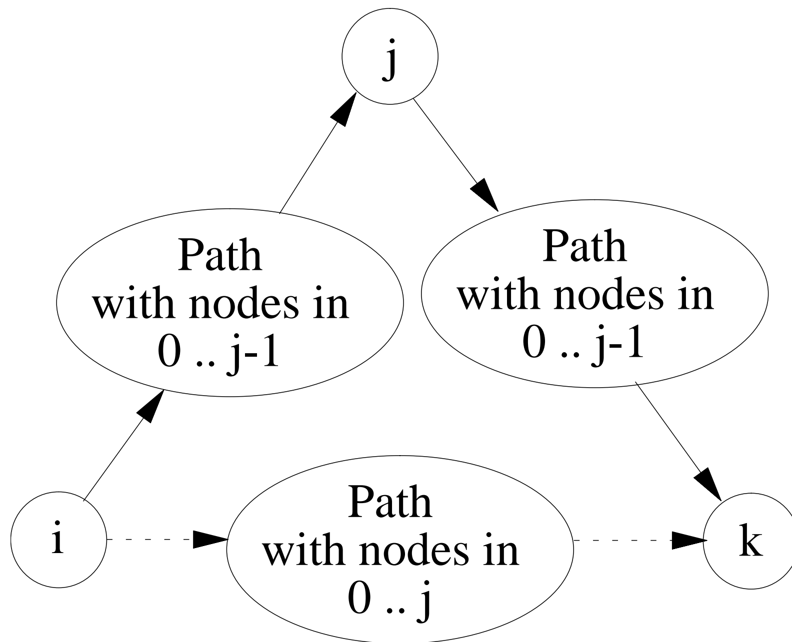
```
barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();

quickSort2(arr,0,arrSize-1);
zeroActive=0;
barrier_wait(&barrier);
printf("CPU used by thread 0 is %f\n",CPUtime()-startCPU);
printf("Elapsed time for thread 0 is %f\n",elapsedTime()-startTime);
barrier_wait(&barrier);
}

void *quickSort1(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
float startCPU,startTime;

barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
while (zeroActive || partitionAvailable)
  if (partitionAvailable)
  {
    quickSort(arr,leftEnd,rightEnd);
    partitionAvailable=0;
  }
barrier_wait(&barrier);
printf("CPU used by thread 1 is %f\n",CPUtime()-startCPU);
printf("Elapsed time for thread 1 is %f\n",elapsedTime()-startTime);
barrier_wait(&barrier);
}

main()
{
int i;
unsigned seed;
thread_data_t *threadWork;
pthread_t thread;
int status;

printf("enter number of keys\n");
scanf("%d",&arrSize);
arr=(int*) malloc(sizeof(int)*arrSize);
if (!arr)
{
  printf("malloc failed\n");
  abort();
}
printf("enter seed\n");
scanf("%u",&seed);
srandom(seed);
srandom(seed);
for (i=0;i<arrSize;i++)
  arr[i]=random();

numThreads=2;
zeroActive=1;
if (barrier_init (&barrier, numThreads))
  printf("barrier_init problem\n");
threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=1;
status = pthread_create (
              &thread, NULL, quickSort1, (void*) threadWork);
if (status != 0)
{
  printf("failed to create thread\n");
  exit(0);
}

threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
threadWork->threadId=0;

maxThreshold=arrSize/4;
```

```
quickSort0((void*) threadWork);
}

[bob@ketchup NOTES02]$ qsort
enter number of keys
3000000
enter seed
432
CPU used is 5.270000
Elapsed time is 5.250000
[bob@ketchup NOTES02]$ qsortPT
enter number of keys
3000000
enter seed
432
CPU used by thread 1 is 3.070000
CPU used by thread 0 is 2.920000
Elapsed time for thread 1 is 3.000000
Elapsed time for thread 0 is 3.000000
```

Example/Warshall's Directed Graph Reachability

WARNING: It is very useful to review the correctness ideas behind Warshall's algorithm before continuing.






```
$ cat warshallPT.c
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "barrier.h"

barrier_t barrier;

int numThreads;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

typedef struct thread_data {
  int threadId;
  int n, **succ;
} thread_data_t;

int **succ,**succSeq,**succInt,**succContig,**succDynamic,n;

allocate(size1, size2, matrix)
int size1,size2;
int ***matrix;
{
  int i;

  if (!(*matrix = (int **) malloc(size1*sizeof(int *))))
    {
      printf("Allocate failed\n");
      exit(2);
    }
  for(i = 0; i < size1; i++)
    if (!((*matrix)[i]=(int *) malloc(size2*sizeof(int))))
    {
      printf("Allocate failed\n");
      exit(3);
    }
}

deallocate(size1,matrix)
int size1;
int **matrix;
{
  int i;

  for(i = 0; i<size1; i++)
    free(matrix[i]);
```

```
  free(matrix);
}

#define generateRandom(minRange,maxRange) \
  (minRange)+abs(random()) % ((maxRange)-(minRange)+1)

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void warshallSeq(n,succ)
int n,**succ;
{
int i,j,k;
float startCPU,startTime;

startCPU=CPUtime();
startTime=elapsedTime();
for (j=0;j<n;j++)
  for (i=0;i<n;i++)
    if (succ[i][j]!=(-1))
      for (k=0;k<n;k++)
        if (succ[i][k]==(-1) && succ[j][k]!=(-1))
          succ[i][k]=succ[i][j];
printf("CPU used by sequential is %f\n",CPUtime()-startCPU);
printf("Elapsed time for sequential is %f\n",elapsedTime()-startTime);
}

void *warshallInt(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int n,**succ;
int i,j,k;
int threadId;
float startCPU,startTime;

threadId=threadWork->threadId;
n=threadWork->n;
succ=threadWork->succ;
barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
for (j=0;j<n;j++)
{
  for (i=threadId;i<n;i+=numThreads)
    if (succ[i][j]!=(-1))
      for (k=0;k<n;k++)
        if (succ[i][k]==(-1) && succ[j][k]!=(-1))
          succ[i][k]=succ[i][j];
  barrier_wait(&barrier);
}
printf("%d: CPU used by interleaved is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("%d: Elapsed time for interleaved is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
free(threadWork);
barrier_wait(&barrier);
}

void *warshallCont(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int n,**succ;
int i,j,k;
int threadId;
```

```
int lower,upper;
float startCPU,startTime;

threadId=threadWork->threadId;
n=threadWork->n;
succ=threadWork->succ;
lower=threadId*n/numThreads;
upper=(threadId+1)*n/numThreads;
barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
for (j=0;j<n;j++)
{
  for (i=lower;i<upper;i++)
    if (succ[i][j]!=(-1))
      for (k=0;k<n;k++)
        if (succ[i][k]==(-1) && succ[j][k]!=(-1))
          succ[i][k]=succ[i][j];
  barrier_wait(&barrier);
}
printf("%d: CPU used by contiguous is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("%d: Elapsed time for contiguous is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
free(threadWork);
barrier_wait(&barrier);
}

int sharedI;
pthread_mutex_t sharedIlock=PTHREAD_MUTEX_INITIALIZER;

void *warshallDynamic(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int n,**succ;
int i,j,k;
int threadId;
float startCPU,startTime;

threadId=threadWork->threadId;
n=threadWork->n;
succ=threadWork->succ;
barrier_wait(&barrier);
startCPU=CPUtime();
startTime=elapsedTime();
for (j=0;j<n;j++)
{
  if (!threadId)
    i=sharedI=0;
  barrier_wait(&barrier);
  if (threadId)
  {
    pthread_mutex_lock(&sharedIlock);
    i= ++sharedI;
    pthread_mutex_unlock(&sharedIlock);
  }
  while (i<n)
  {
    if (succ[i][j]!=(-1))
      for (k=0;k<n;k++)
        if (succ[i][k]==(-1) && succ[j][k]!=(-1))
          succ[i][k]=succ[i][j];
    pthread_mutex_lock(&sharedIlock);
    i= ++sharedI;
    pthread_mutex_unlock(&sharedIlock);
  }
  barrier_wait(&barrier);
}
printf("%d: CPU used by dynamic is %f\n",threadWork->threadId,
  CPUtime()-startCPU);
printf("%d: Elapsed time for dynamic is %f\n",threadWork->threadId,
  elapsedTime()-startTime);
free(threadWork);
barrier_wait(&barrier);
```

```
}

void printPaths(n,succ)
int n,**succ;
{
int i,j,k;

for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    if (succ[i][j]==(-1))
      printf("No path from %d to %d\n",i,j);
    else
    {
      printf("Path %d ",i);
      k=succ[i][j];
      while (k!=j)
      {
        printf("%d ",k);
        k=succ[k][j];
      }
      printf("%d\n",j);
    }

}

main()
{
int i,j,k,seed,status;
int tail,head;
thread_data_t *threadWork;
pthread_t thread;

printf("Enter # of vertices\n");
scanf("%d",&n);
allocate(n,n,&succ);
allocate(n,n,&succSeq);
allocate(n,n,&succInt);
allocate(n,n,&succContig);
allocate(n,n,&succDynamic);
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    succ[i][j]=(-1);
printf("Enter seed\n");
scanf("%d",&seed);
srandom(seed);
for (i=0;i<n/2;i++)
  for (j=0;j<n/2;j++)
  {
    tail=generateRandom(0,n-1);
    head=generateRandom(0,n-1);
    succ[tail][head]=head;
  }
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    succSeq[i][j]=succ[i][j];
warshallSeq(n,succSeq);

for (numThreads=2;numThreads<=2;numThreads++)
{
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      succInt[i][j]=succContig[i][j]=succDynamic[i][j]=succ[i][j];
  printf("Using %d threads\n",numThreads);
  if (barrier_init (&barrier, numThreads))
    printf("barrier_init problem\n");

  for (i=1;i<numThreads;i++)
  {
    threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
    threadWork->threadId=i;
    threadWork->n=n;
    threadWork->succ=succInt;
    status = pthread_create (&thread,NULL,warshallInt,threadWork);
    if (status != 0)
```

```
    {
      printf("failed to create thread\n");
      exit(0);
    }
  }
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=0;
  threadWork->n=n;
  threadWork->succ=succInt;
  warshallInt(threadWork);

  for (i=1;i<numThreads;i++)
  {
    threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
    threadWork->threadId=i;
    threadWork->n=n;
    threadWork->succ=succContig;
    status = pthread_create (&thread,NULL,warshallCont,threadWork);
    if (status != 0)
    {
      printf("failed to create thread\n");
      exit(0);
    }
  }
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=0;
  threadWork->n=n;
  threadWork->succ=succContig;
  warshallCont(threadWork);

  for (i=1;i<numThreads;i++)
  {
    threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
    threadWork->threadId=i;
    threadWork->n=n;
    threadWork->succ=succDynamic;
    status = pthread_create (&thread,NULL,warshallDynamic,threadWork);
    if (status != 0)
    {
      printf("failed to create thread\n");
      exit(0);
    }
  }
  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
  threadWork->threadId=0;
  threadWork->n=n;
  threadWork->succ=succDynamic;
  warshallDynamic(threadWork);

  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      if (succSeq[i][j]!=succInt[i][j] ||
      succSeq[i][j]!=succContig[i][j] ||
      succSeq[i][j]!=succDynamic[i][j])
        printf("[%d][%d]: %d %d %d %d\n",i,j,
          succSeq[i][j],succInt[i][j],succContig[i][j],
          succDynamic[i][j]);
}
deallocate(n,succ);
deallocate(n,succSeq);
deallocate(n,succInt);
deallocate(n,succContig);
deallocate(n,succDynamic);
}

[bob@ketchup NOTES02]$ warshallPT
Enter # of vertices
500
Enter seed
345
CPU used by sequential is 5.290000
Elapsed time for sequential is 5.500000
Using 2 threads
1: CPU used by interleaved is 2.470000
```

```
0: CPU used by interleaved is 2.490000
1: Elapsed time for interleaved is 2.500000
0: Elapsed time for interleaved is 2.500000
1: CPU used by contiguous is 2.460000
0: CPU used by contiguous is 2.469999
1: Elapsed time for contiguous is 2.500000
0: Elapsed time for contiguous is 2.500000
1: CPU used by dynamic is 2.830000
0: CPU used by dynamic is 2.860001
1: Elapsed time for dynamic is 2.750000
0: Elapsed time for dynamic is 2.750000
```

Contiguous Allocation for Enumeration Problems

CONCEPT - Ranking and unranking of combinatorial objects

Note: To be compatible with coding in C, we use rank = 0 for the "first" object instead of rank = 1. In addition, we enumerate using 0 .. n-1 instead of 1 .. n.

Very simple example - powerset enumeration (`powersetEnum.c`)

```
/* Enumeration of powerset on base set with n elements */
/* Elements are from 0 .. n-1 */
/* Ranks are from 0 .. 2**n - 1 */
/* Processing assumes that subset arrays are sorted
   with -1 as a terminator, but input arrays do not
   have to be sorted */

#include <stdio.h>

int n,count;

int cardPowerset(n)
int n;
{
return 1<<n;  /* 2**n */
}

void rank2subset(rank,subset)
int rank,subset[20];
{
int i,j,bit;

j=0;
for (i=0;i<n;i++)
{
  bit=rank & 1;
  if (bit)
  {
    subset[j]=i;
    j++;
  }
  rank=rank>>1;
}
subset[j]=(-1);
}

void readSubset(subset)
int subset[20];
{
int i;

while (1)
{
  printf("enter subset elements with -1 to terminate\n");
  for (i=0;;i++)
  {
    scanf("%d",&subset[i]);
    if (subset[i]==(-1))
      break;
    if (subset[i]<0 || subset[i]>=n)
    {
```

```
      printf("bad input\n");
      break;
    }
  }
  if (subset[i]==(-1))
    break;
}
}

void printSubset(subset)
int subset[20];
{
int i;

printf("{");
for (i=0;subset[i]!=(-1);i++)
{
  printf("%d",subset[i]);
  if (subset[i+1]!=(-1))
    printf(",");
}
printf("}\n");
}

int subset2rank(subset)
int subset[20];
{
int i,rank;

rank=0;
for (i=0;subset[i]!=(-1);i++)
  rank=rank | (1<<subset[i]);
return rank;
}

void nextSubset(subsetA,subsetB)
int subsetA[20],subsetB[20];  /*Input & output arrays*/
{
int i,j;

for (i=0;subsetA[i]==i;i++)
  ;
subsetB[0]=i;
for (j=1;subsetA[i]!=(-1);)
{
  subsetB[j]=subsetA[i];
  i++;
  j++;
}
subsetB[j]=(-1);
}

main()
{
int rank,numSubsets,i;
int subset[20],subset2[20];
int *subsetA,*subsetB,*temp;

printf("enter n\n");
scanf("%d",&n);
count=cardPowerset(n);
printf("rank -> subset\n");
while (1)
{
  printf("enter rank (-1 to terminate)\n");
  scanf("%d",&rank);
  if (rank<(-1) || rank>=count)
  {
    printf("bad rank\n");
    continue;
  }
  if (rank==(-1))
    break;
  rank2subset(rank,subset);
```

```
  printSubset(subset);
}
printf("subset -> rank\n");
while (1)
{
  printf("wish to convert a case? (1=yes 0=no)\n");
  scanf("%d",&i);
  if (i<0 || i>1)
  {
    printf("bad response\n");
    continue;
  }
  if (!i)
    break;
  readSubset(subset);
  printf("rank is %d\n",subset2rank(subset));
}
printf("enumerate subsets starting from a rank\n");
while (1)
{
  printf("enter rank & number of subsets (-1 -1 to stop)\n");
  scanf("%d %d",&rank,&numSubsets);
  if (rank==numSubsets && rank==(-1))
    break;
  if (rank<0 || rank>=count)
  {
    printf("bad rank\n");
    continue;
  }
  if (numSubsets<1 || rank+numSubsets-1>=count)
  {
    printf("bad number of subsets\n");
    continue;
  }
  rank2subset(rank,subset);
  printf("starting subset: ");
  subsetB=subset;
  subsetA=subset2;
  for (i=0;i<numSubsets-1;i++)
  {
    printSubset(subsetB);
    temp=subsetA;
    subsetA=subsetB;
    subsetB=temp;
    nextSubset(subsetA,subsetB);
  }
  printSubset(subsetB);
}
}
```

ENUMERATING PERMUTATIONS

k = positions

n = available objects (from the set {0, . . ., n-1})

Let k=4, n=8. Number of permutations = n!/(n - k)! = 8!/(8 - 4)! = 8*7*6*5

Lexicographic Ordering = String Comparison Applied to Permutations

Generating permutations is often done in lexicographic ordering

Principle: Maintain pool of available objects. Replace low-order position by the next higher available object. If none available, release object in next column over and attempt to replace it with next higher object. After successfully replacing an object, replace objects in low-order positions using the lowest objects available.

Permutation	Pool
2 5 7 0	1 3 4 6
2 5 7 1	0 3 4 6
2 5 7 3	0 1 4 6

2 5 7 4	0 1 3 6
2 5 7 6	0 1 3 4
2 6 0 1	3 4 5 7
2 6 0 3	1 4 5 7
2 6 0 4	1 3 5 7
2 6 0 5	1 3 4 7
2 6 0 7	1 3 4 5
2 6 1 0	3 4 5 7

Under this ordering, 0 1 2 3 is the first permutation and 7 6 5 4 is the last permutation.

NUMBERING PERMUTATIONS

Based on a ‘‘variable radix’’ numbering system. Allows us to number consecutively (‘‘rank’’) the permutations generated by the preceding algorithm.

Let n=6 and k=4. Let p_i be the object in the i-th column (left-to-right).

$$r_i = p_i - i + \sum_{j=0}^{i-1} p_i < p_j$$

is the position of p_i within the radix for that column. This is based on the first column having n choices, the second column having n - 1 choices, ... column i having n + 1 - i choices for its value, since earlier columns have removed one object each.

The summation counts the number of previous positions with larger values.

Suppose that the permutation is 4 2 3 1:

	0	1	2	3
p	4	2	3	1
r	4	2	2	1

Position in column:	0	1	2	3	4	5
$r_0 = 4 - 0 + 0 = 4$	0	1	2	3	**4**	5
$r_1 = 2 - 1 + 1 = 2$	0	1	**2**	3	5	
$r_2 = 3 - 2 + 1 = 2$	0	1	**3**	5		
$r_3 = 1 - 3 + 3 = 1$	0	**1**	5			

Column Values (Radix-Based)

	0	1	2	3
radix	6	5	4	3
r	4	2	2	1
column value	60	12	3	1
rank	4*60 +	2*12 +	2*3 +	1*1
	240 +	24 +	6 +	1 = 271

Under this scheme, permutation 0 1 2 3 would have rank 0 and permutation 5 4 3 2 would have rank 359.

Given the rank (d), the permutation may be found:

$$r_i = \frac{d - \sum_{j=0}^{i-1} r_j * \text{column value}_j}{\text{column value}_i}$$

From the preceding example, d = 271

$r_0 = 271/60 = 4$

$r_1 = (271 - 4*60)/12 = 31/12 = 2$

$r_2 = (271 - 4*60 - 2*12)/3 = 7/3 = 2$

$r_3 = (271 - 4*60 - 2*12 - 2*3)/1 = 1/1 = 1$

$p_i = r_i + i - d_i$, where d_i is the smallest integer such that

$$d_i = \sum_{0}^{i-1} r_i + 1 - d_i < p_j$$

which means the i+1st available integer.

	0	1	2	3	4	5
$r_0 = 4$, so go over 5 positions					p_0	
$r_1 = 2$, so go over 3 positions			p_1		p_0	
$r_2 = 2$, so go over 3 positions			p_1	p_2	p_0	
$r_3 = 1$, so go over 2 positions		p_3	p_1	p_2	p_0	

Thus giving the correct permutation, 4 2 3 1

Another example: k=4, n = 8, permutation = 5 1 6 2

	0	1	2	3
p	5	1	6	2
r	5	1	4	1

Position in column:	0	1	2	3	4	5	6	7
$r_0 = 5 - 0 + 0 = 5$	0	1	2	3	4	**5**	6	7
$r_1 = 1 - 1 + 1 = 1$	0	**1**	2	3	4	6	7	
$r_2 = 6 - 2 + 0 = 4$	0	2	3	4	**6**	7		
$r_3 = 2 - 3 + 2 = 1$	0	**2**	3	4	7			

Column Values

	0		1		2		3	
radix	8		7		6		5	
r	5		1		4		1	
column value	210		30		5		1	
rank	5*210	+	1*30	+	4*5	+	1*1	
	1050	+	30	+	20	+	1	= 1101

Permutation from rank d = 1101

$r_0 = 1101/210 = 5$

$r_1 = 51/30 = 1$

$r_2 = 21/5 = 4$

$r_3 = 1/1 = 1$

	0	1	2	3	4	5	6	7
$r_0 = 5$, so go over 6 positions						p_0		
$r_1 = 1$, so go over 2 positions		p_1				p_0		
$r_2 = 4$, so go over 5 positions		p_1				p_0	p_2	
$r_3 = 1$, so go over 2 positions		p_1	p_3			p_0	p_2	

Giving the permutation 5 1 6 2

`permEnum.c`:

```
/* Enumeration of k permutations on base set with n elements */
/* Elements are from 0 .. n-1 */
/* Ranks are from 0 .. n*(n-1)*...*(n-k+1) - 1 */


#include <stdio.h>

int n,k,count;
int columnValue[20];

int cardPermset(n,k)
int n,k;
{
int i,card;

columnValue[k-1]=1;
for (i=k-2;i>=0;i--)
  columnValue[i]=columnValue[i+1]*(n-i-1);
card=n*columnValue[0];
printf("column values: ");
for (i=0;i<k;i++)
  printf("%d ",columnValue[i]);
printf("\n");
printf("cardinality of permutation set %d\n",card);
return card;
}

void rank2perm(rank,perm)
int rank,perm[20];
{
int i,j;
int r[20],available[20];

for (i=0;i<k;i++)
{
  r[i]=rank/columnValue[i];
  rank=rank%columnValue[i];
}

for (i=0;i<n;i++)
  available[i]=i;
for (i=0;i<k;i++)
{
  perm[i]=available[r[i]];
  for (j=r[i];j<n-1-i;j++)
    available[j]=available[j+1];
}
}

void readPerm(perm)
int perm[20];
{
int i;
```

```
int alreadyRead[20];

while (1)
{
  for (i=0;i<n;i++)
    alreadyRead[i]=0;
  printf("enter %d permutation elements\n",k);
  for (i=0;i<k;i++)
  {
    scanf("%d",&perm[i]);
    if (perm[i]<0 || perm[i]>=n || alreadyRead[perm[i]])
    {
      printf("bad input\n");
      break;
    }
    alreadyRead[perm[i]]=1;
  }
  if (i==k)
    break;
}
}

void printPerm(perm)
int perm[20];
{
int i;

for (i=0;i<k;i++)
  printf("%d ",perm[i]);
printf("\n");
}

int perm2rank(perm)
int perm[20];
{
int i,j,rank;
int available[20],r[20];

for (i=0;i<n;i++)
  available[i]=i;
for (i=0;i<k;i++)
{
  for (j=0;perm[i]!=available[j];j++)
    ;
  r[i]=j;
  for (;j<n-1-i;j++)
    available[j]=available[j+1];
}
rank=0;
for (i=0;i<k;i++)
  rank+=r[i]*columnValue[i];
return rank;
}

void nextPerm(permA,poolA,permB,poolB)
int permA[20],poolA[20],permB[20],poolB[20];  /*Input & output arrays*/
{
/*This could be faster*/
int i,j;

for (i=0;i<k;i++)
  permB[i]=permA[i];
for (i=0;i<n;i++)
  poolB[i]=poolA[i];
for (i=k-1;;i--)
{
  poolB[permB[i]]=1;
  for (j=permB[i]+1;j<n && !poolB[j];j++)
    ;
  if (j<n)
    break;
}
permB[i]=j;
poolB[j]=0;
```

```
j=0;
for (i++;i<k;i++)
{
  for (;!poolB[j];j++)
    ;
  permB[i]=j;
  poolB[j]=0;
  j++;
}
return;
}

void perm2pool(perm,pool)
int perm[20],pool[20];
{
int i;

for (i=0;i<n;i++)
  pool[i]=1;
for (i=0;i<k;i++)
  pool[perm[i]]=0;
}

main()
{
int rank,numPerms,i;
int perm[20],pool[20],perm2[20],pool2[20];
int *permA,*poolA,*permB,*poolB,*tempPerm,*tempPool;printf("enter n and k\n");
scanf("%d %d",&n,&k);
count=cardPermset(n,k);
printf("rank -> perm\n");
while (1)
{
  printf("enter rank (-1 to terminate)\n");
  scanf("%d",&rank);
  if (rank<(-1) || rank>=count)
  {
    printf("bad rank\n");
    continue;
  }
  if (rank==(-1))
    break;
  rank2perm(rank,perm);
  printPerm(perm);
}
printf("perm -> rank\n");
while (1)
{
  printf("wish to convert a case? (1=yes 0=no)\n");
  scanf("%d",&i);
  if (i<0 || i>1)
  {
    printf("bad response\n");
    continue;
  }
  if (!i)
    break;
  readPerm(perm);
  printf("rank is %d\n",perm2rank(perm));
}
printf("enumerate permutations starting from a rank\n");
while (1)
{
  printf("enter rank & number of permutations (-1 -1 to stop)\n");
  scanf("%d %d",&rank,&numPerms);
  if (rank==numPerms && rank==(-1))
    break;
  if (rank<0 || rank>=count)
  {
    printf("bad rank\n");
    continue;
  }
  if (numPerms<1 || rank+numPerms-1>=count)
  {
```

```
    printf("bad number of permutations\n");
    continue;
  }
  rank2perm(rank,perm);
  perm2pool(perm,pool);
  printf("starting permutation: ");
  permB=perm;
  poolB=pool;
  permA=perm2;
  poolA=pool2;
  for (i=0;i<numPerms-1;i++)
  {
    printPerm(permB);
    tempPerm=permA;
    tempPool=poolA;
    permA=permB;
    poolA=poolB;
    permB=tempPerm;
    poolB=tempPool;
    nextPerm(permA,poolA,permB,poolB);
  }
  printPerm(permB);
}
}
```

ENUMERATING COMBINATIONS

Ignore ordering in sequence, unlike permutations

k = size of set

n = number of available objects (set is 0 .. n - 1)

Let k = 3, n = 9. Number of combinations $= C(9,3) = \binom{9}{3} = \frac{n!}{(n-k)!k!} = 84$

Generating in lexicographic order: ONLY WANT PERMUTATIONS THAT ARE SORTED!!!

Observation: The last combination to be generated is (n - k) . . . (n-2)(n-1), e.g. $(c_0\ c_1\ c_2) = 6\ 7\ 8$ when k=3, n=9. In each of the k positions for a valid combination, the value will never exceed that in the respective position for the last combination.

Suppose that j is the rightmost position for which $c_j < n - k + j$, then the next combination is found by

1. Incrementing c_j by 1

2. Replace c_{j+1} ... c_{k-1} by successive values

Example: n=8, k=4 and the most recent permutation generated is $(c_0\ c_1\ c_2\ c_3) = 2\ 4\ 6\ 7$

7 cannot be the j position, since we don’t have 7 < 8 - 4 + 3

6 cannot be the j position, since we don’t have 6 < 8 - 4 + 2

4 can be replaced, since 4 < 8 - 4 + 1

This gives 2 5 6 7 (and then 3 4 5 6, 3 4 5 7, 3 4 6 7, 3 5 6 7 . . .)

NUMBERING COMBINATIONS

To demonstrate the method, suppose that n=5 and k=3. There are 10 combinations (lexicographically) by the preceeding procedure:

0 1 2

0 1 3
0 1 4
0 2 3 - - Determine the rank of this combination, i.e. 3
0 2 4
0 3 4
1 2 3
1 2 4
1 3 4
2 3 4

Ranking is based on the complement of a combination (C').

1. Reverse order of positions
2. Subtract each position from n - 1.

Suppose that n=5, k=3. (0 2 3)' = (1 2 4)

$$\text{Rank of combination C} = \binom{n}{k} - \sum_{i=0}^{k-1} \binom{c'_i}{i+1} - 1$$

Concept: Subtract counts for combinations that are after this one

$$\text{Rank of } (0 \quad 2 \quad 3) = \binom{5}{3} - \binom{1}{1} - \binom{2}{2} - \binom{4}{3} - 1 = 10 - 1 - 1 - 4 - 1 = 3$$

using (1 2 4) to get 10 total combinations - 1 way to change only the last position - 1 way to change only the last two positions and - 4 ways to change the last three positions - 1 to ensure that first combination has rank 0, so this combination must be ranked 3rd.

Recovering the combination from its rank:

1. Get back to complement (the hard part).
2. Complement the complement (the easy part).

Using rank = 3 from previous example, we can easily determine that the 3 subtracted counts of combinations add up to 6. Most of these combinations must come from varying all three positions:

Find c'_2: $6 \geq C(j,3)$ For j = 4, C(4,3) = 4 so $c'_2 = 4$.

Find c'_1: $6 - 4 = 2 \geq C(j,2)$ For j = 3, C(3,2) = 3 (problem with $\geq$), try j = 2, C(2,2) = 1, so $c'_1 = 2$

Find c'_0: $2 - 1 = 1 \geq C(j,1))$ For j = 1, C(1,1)) = 1, so $c'_0 = 1$

The original combination comes from $(c'_0\ c'_1\ c'_2)'$ = (1 2 4)'= (0 2 3)

Another example: n=10, k=4, C(10,4) = 210, combination is $(c_0\ c_1\ c_2\ c_3)$ = 0 2 6 8

(0 2 6 8)' = 1 3 7 9

Rank = 210 - C(1,1) - C(3,2) - C(7,3) - C(9,4) - 1 = 210 - 1 - 3 - 35 - 126 - 1 = 44

This uses reasoning analogous to before:

C(1,1) corresponds to changing 8 in (0 2 6 8) to 9.

C(3,2) corresponds to changing 6 8 in (0 2 6 8) to any pair from {7, 8, 9}

C(7,3) corresponds to changing 2 6 8 in (0 2 6 8) to any triple from {3, 4, 5, 6, 7, 8, 9}

C(9,4) corresponds to changing all positions to any quadruple from {1, 2, 3, 4, 5, 6, 7, 8, 9}

1 corresponds to ranking the first combination as zero instead of one.

Recovering combination from rank (same example):

210 - 1 - 44 = 165

Find c'_3: $165 \geq C(j,4)$ For j=9, C(9,4) = 126, so $c'_3 = 9$

Find c'_2: $165 - 126 = 39 \geq C(j,3)$ For j=8, C(8,3) = 56, For j=7, C(7,3) = 35, so $c'_2 = 7$

Find c'_1: $39 - 35 = 4 \geq C(j,2)$ For j=6, C(6,2) = 15, For j=5, C(5,2) = 10,
For j=4, C(4,2) = 6, For j=3, C(3,2) = 3, so $c'_1 = 3$

Find c'_0: $4 - 3 = 1 \geq C(j,1)$ For j=2, C(2,1) = 2, For j=1, C(1,1) = 1, so $c'_0 = 1$

The original combination comes from $(c'_0\ c'_1\ c'_2\ c'_3)'$ = (1 3 7 9)' = (0 2 6 8)

combEnum.c

```
/* Enumeration of k combinations on base set with n elements */
/* Elements are from 0 .. n-1 */
/* Ranks are from 0 .. C(n,k) - 1 */


#include <stdio.h>

int n,k,count;
int C[20][20];

int cardCombset(n,k)
int n,k;
{
int i,j,card;

for (i=0;i<=n;i++)
{
  C[i][0]=1;
  C[i][i]=1;
}
for (i=2;i<=n;i++)
  for (j=1;j<i;j++)
    C[i][j]=C[i-1][j-1]+C[i-1][j];
card=C[n][k];
printf("cardinality of combination set %d\n",card);
return card;
}

void rank2comb(rank,comb)
int rank,comb[20];
{
int i,j,leftover;
int complement[20];

leftover=count-rank-1;
j=n-1;
for (i=k-1;i>=0;i--)
{
  for (;leftover<C[j][i+1];j--)
    ;
  complement[i]=j;
  leftover-=C[j][i+1];
  j--;
}
```

```
for (i=0;i<k;i++)
  comb[i]=n-1-complement[k-1-i];
}

void readComb(comb)
int comb[20];
{
int i;

while (1)
{
  printf("enter %d combination elements\n",k);
  for (i=0;i<k;i++)
  {
    scanf("%d",&comb[i]);
    if (comb[i]<0 || comb[i]>=n)
    {
      printf("bad input\n");
      break;
    }
  }
  if (i==k)
  {
    for (i=0;i<k-1;i++)
      if (comb[i]>=comb[i+1])
        break;
    if (i==k-1)
      break;
    printf("bad input\n");
  }
}
}

void printComb(comb)
int comb[20];
{
int i;

for (i=0;i<k;i++)
  printf("%d ",comb[i]);
printf("\n");
}

int comb2rank(comb)
int comb[20];
{
int i,j,rank;
int complement[20];

for (i=0;i<k;i++)
  complement[i]=n-1-comb[k-1-i];
rank=C[n][k];
for (i=0;i<k;i++)
  rank-=C[complement[i]][i+1];
rank--;
return rank;
}

void nextComb(combA,combB)
int combA[20],combB[20];  /*Input & output arrays*/
{
int i,j;

for (i=0;i<k;i++)
  combB[i]=combA[i];
for (j=k-1;combB[j]==n-k+j;j--)
  ;
combB[j]++;
for (j++;j<k;j++)
  combB[j]=combB[j-1]+1;
}

main()
{
```

```
int rank,numCombs,i;
int comb[20],comb2[20];
int *combA,*combB,*temp;

printf("enter n and k\n");
scanf("%d %d",&n,&k);
count=cardCombset(n,k);
printf("rank -> comb\n");
while (1)
{
  printf("enter rank (-1 to terminate)\n");
  scanf("%d",&rank);
  if (rank<(-1) || rank>=count)
  {
    printf("bad rank\n");
    continue;
  }
  if (rank==(-1))
    break;
  rank2comb(rank,comb);
  printComb(comb);
}
printf("comb -> rank\n");
while (1)
{
  printf("wish to convert a case? (1=yes 0=no)\n");
  scanf("%d",&i);
  if (i<0 || i>1)
  {
    printf("bad response\n");
    continue;
  }
  if (!i)
    break;
  readComb(comb);
  printf("rank is %d\n",comb2rank(comb));
}
printf("enumerate combinations starting from a rank\n");
while (1)
{
  printf("enter rank & number of combinations (-1 -1 to stop)\n");
  scanf("%d %d",&rank,&numCombs);
  if (rank==numCombs && rank==(-1))
    break;
  if (rank<0 || rank>=count)
  {
    printf("bad rank\n");
    continue;
  }
  if (numCombs<1 || rank+numCombs-1>=count)
  {
    printf("bad number of combinations\n");
    continue;
  }
  rank2comb(rank,comb);
  printf("starting combination: ");
  combB=comb;
  combA=comb2;
  for (i=0;i<numCombs-1;i++)
  {
    printComb(combB);
    temp=combA;
    combA=combB;
    combB=temp;
    nextComb(combA,combB);
  }
  printComb(combB);
}
}
```

Suppose P processors are to compute all permutations/combinations. Each processor generates a fraction of the codes in contiguous fashion:

1. Compute number of codes N.
2. If processors are numbered 0 .. P-1, processor i takes codes from rank = iN/P through rank = (i+1)N/P - 1.
3. Conversion from rank to permutation/combination is performed ONCE by each processor, consecutively ranked permutations are generated using the "next" algorithms.
4. Gives linear speed-up.

Note: If ranking is difficult, then dynamic scheduling may be necessary.

An old lab assignment involving parallelism based on combinations.

Goals:

1. Write two parallel pthread programs for the traveling salesperson problem. Sequential code for an exhaustive (dynamic programming) search has been provided (`tsp2.c`).

2. Evaluate the obtained parallelism using ‘‘ptime’’ (on the shell command line) and getrusage() calls in the thread function.

Requirements:

1. Submit a listing of a parallel program based on interleaving the loop `for (j=0;j<i;j++)` in `solveTSP()`.

2. Submit a listing of a parallel program based on the dynamic scheduling approach that allocates subsets of the vertices to a processor. This approach is used for the loop `while (combBits)` in `solveTSP()`.

3. Run your programs with 2 threads on various inputs. Provide execution output.

4. Submit a brief report comparing your programs and the original sequential program.

Getting Started:

1. Prepare a small input file (4 vertices), compile `tsp2.c` (edit to include debugging `printf`'s), and execute. Observe the optimal path produced for each combination of vertices along with a designated "end" vertex. Note that paths with k vertices are produced before those with k + 1 vertices.

2. Compile `tsp2.c` without debugging `printf`'s. Time some executions.

3. Understand `tsp2.c`. In particular, observe the following:

   a. The input edge weights are stored as an adjacency matrix.

   b. Subscripting for `TSPtable`. Suppose that we are working with |V| = 10. Thus, each path will begin with |V| - 1 = 9. Now consider the optimal path that also includes vertices 2, 4, 5, 7 and has 4 as its end. This path would be found at TSPtable[180][1], since $180 = 2^2 + 2^4 + 2^5 + 2^7$ and vertex 4 has the second smallest label among the included vertices for this path (if vertex 7 were the end, then `TSPtable[180][3]` would be used). The pathStruct for this `TSPtable` entry provides 1) the length (cost) of the optimal path and 2) the vertex (`nextToLastVertex`) that is the predecessor of the last vertex on the optimal path.

   c. The allocation for `TSPtable`. Even though the storage is `malloc`'ed early in the execution, storage is commited throughout the dynamic programming algorithm.

   d. Observe how combinations of vertices get generated by `firstCombination()` and `nextCombination()`. In your first version, all processes work together in the dynamic programming for one combination at a time. In your second version, only one process will work on each combination.

4. Convert the code to get the first version. It is easier to have all processes redundantly execute `firstCombination()` and `nextCombination()`.

5. Convert the code to get the second version. `firstCombination()` and `nextCombination()` will need to be modified to save additional state information to facilitate the worklist approach. Notes 12 reviews the generation of combinations in lexicographic order.

Input File:

```
5
0 1 1
1 0 2
0 2 3
2 0 2
1 2 2
2 1 1
2 3 1
3 2 2
2 4 2
4 2 3
3 0 5
0 3 4
3 4 3
4 3 4
4 0 1
0 4 1
-1 -1 -1
/usr/faculty/weems: tsp2<dat2
WARNING - missing edge 1->3
WARNING - missing edge 1->4
WARNING - missing edge 3->1
WARNING - missing edge 4->1
DEBUG Cost=1 for path 4 0
DEBUG Cost=999999 for path 4 1
DEBUG Cost=3 for path 4 2
DEBUG Cost=4 for path 4 3
DEBUG Cost=1000001 for path 4 1 0
DEBUG Cost=2 for path 4 0 1
DEBUG Cost=5 for path 4 2 0
DEBUG Cost=4 for path 4 0 2
DEBUG Cost=9 for path 4 3 0
DEBUG Cost=5 for path 4 0 3
DEBUG Cost=4 for path 4 2 1
DEBUG Cost=1000001 for path 4 1 2
DEBUG Cost=1000003 for path 4 3 1
DEBUG Cost=1999998 for path 4 1 3
DEBUG Cost=6 for path 4 3 2
DEBUG Cost=4 for path 4 2 3
DEBUG Cost=6 for path 4 2 1 0
DEBUG Cost=5 for path 4 0 2 1
DEBUG Cost=4 for path 4 0 1 2
DEBUG Cost=1000005 for path 4 3 1 0
DEBUG Cost=10 for path 4 3 0 1
DEBUG Cost=1000001 for path 4 0 1 3
DEBUG Cost=8 for path 4 3 2 0
DEBUG Cost=7 for path 4 0 3 2
DEBUG Cost=5 for path 4 0 2 3
DEBUG Cost=7 for path 4 3 2 1
DEBUG Cost=1000005 for path 4 3 1 2
DEBUG Cost=1000002 for path 4 1 2 3
DEBUG Cost=9 for path 4 3 2 1 0
DEBUG Cost=8 for path 4 0 3 2 1
DEBUG Cost=12 for path 4 3 0 1 2
DEBUG Cost=5 for path 4 0 1 2 3
Cost=8 for path 4 0 1 2 3 4
```

```
[bob@ketchup NOTES12]$ tsp2<tsp21points.dat
i=2
i=3
i=4
i=5
i=6
i=7
i=8
i=9
i=10
i=11
i=12
i=13
```

```
i=14
i=15
i=16
i=17
i=18
i=19
i=20
Cost=48 for path 20 2 15 19 8 10 6 17 12 3 4 9 5 11 16 1 0 14 7 18 13 20
```

SEARCH

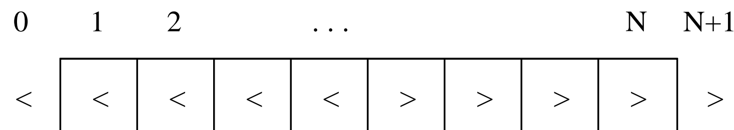
Search of Unordered Array - Not Interesting

Search of Ordered Array on CRCW PRAM (identical write property)

N processors, n keys (N << n)

```
low := 1
high := n
found := 0
compare[0] := '<'
compare[N+1] := '>'
while low ≤ high and found=0 do
    for i:=1 to N do {in parallel}
        mid[i] := low + i(high - low)/(N + 1)
        if A[mid[i]] = key
            found := mid[i]
        else if A[mid[i]] < key
            compare[i] := '<'
        else
            compare[i] := '>'
        endif
        if compare[i] = '<' and compare[i+1] = '>'
            low := mid[i] + 1
        else if compare[i-1] = '<' and compare[i] = '>'
            high := mid[i] - 1
        endif
    endfor
endwhile
```

Correctness: Based on compare array usage



Time is analogous to binary search:

binary search probes = ceiling(lg n) + 1

N-processor search parallel probes = ceiling($\log_{(N+1)}$ n) + 1, which is about lg n / lg(N+1)

Speed-up is about lg N

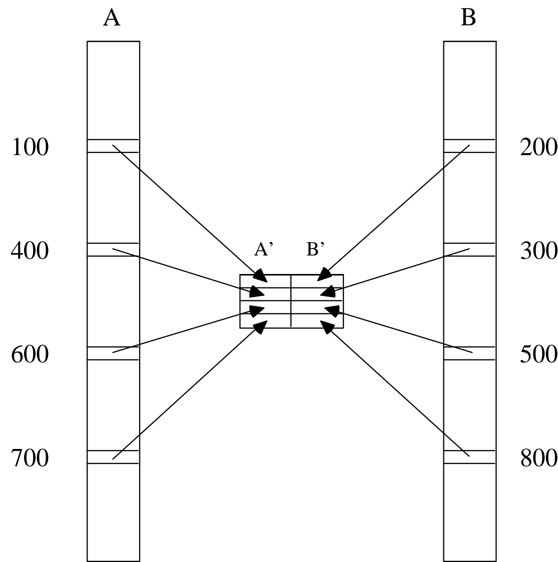
Efficiency = lg N/N

MERGING ORDERED ARRAYS ON CREW PRAM

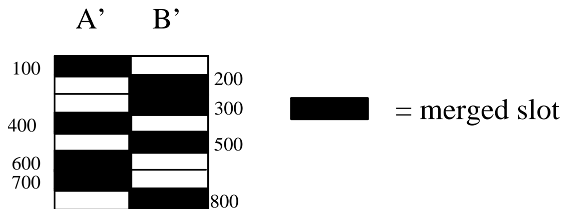
Algorithm Sketch:

N elements in each array, P processors

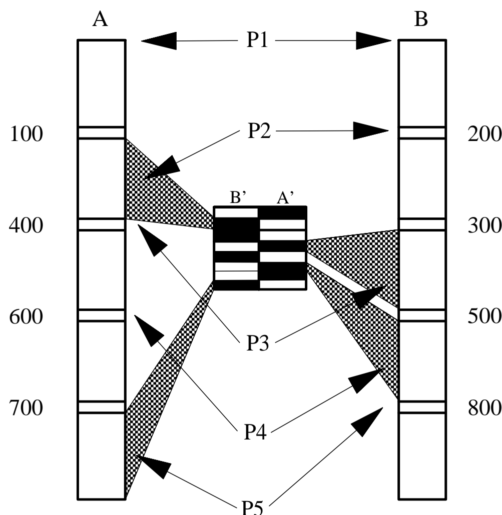
Step 1: Divide the two ordered arrays into P pieces by "sampling" P - 1 evenly spaced slots



Step 2: Merge the A' and B' tables according to the sampled keys:

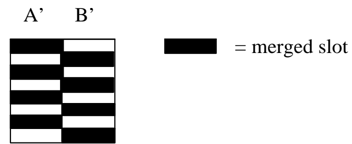


Step 3: Determine where each processor will start merging (array Q). Use the "even" elements from the Step 2 merge and find successor element in merge by using binary search on the opposite arrray. Shaded region indicates range of slots where binary search could stop.

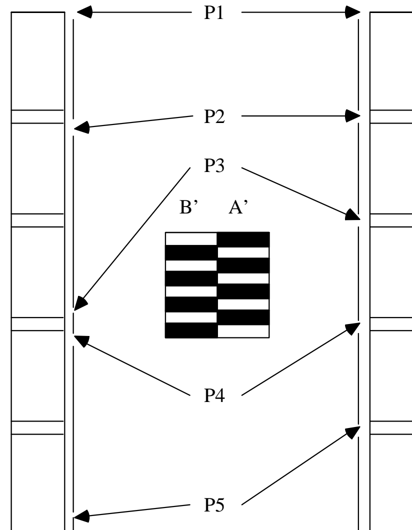


Step 4: Each processor merges elements from start point (indicated in array Q) up to start point for the next processor (again in Q) or the end of the array if the last processor.

CLAIM: No processor must merge "substantially" more than 3N/P array elements nor less than N/P elements, ASSUMING no duplicate keys (which would require more details for the algorithm). Based on our example, worst case occurs with Step 2 as:



Step 3 is now:



crewMerge95PT.c

```
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include "barrier.h"

#define DEBUG 0
#define NUMSPERLINES 10

barrier_t barrier;

int numThreads;

typedef struct thread_data {
  int threadId;
} thread_data_t;

typedef struct {int key,index; char sequenceName;} triple;
triple *v;

typedef struct {int aPos,bPos;} pair;
pair *Q;

int *a,*b,*c;
int arrSize,arrSize2;
int *aPrime,*bPrime;

void primeInit()
{
int i;

for (i=0;i<numThreads;i++)
{
  aPrime[i]=a[i*arrSize/numThreads];
  bPrime[i]=b[i*arrSize/numThreads];
}
}

int binSearch(key,arrPoint,lower,upper)
int key,*arrPoint,lower,upper;
{
```

```
int low,high,mid;

low=lower;
high=upper;
while (low<=high)
{
  mid=(low+high)/2;
  if (key==arrPoint[mid])
    break;
  if (key<arrPoint[mid])
    high=mid-1;
  else
    low=mid+1;
}
while (mid<=upper && key>=arrPoint[mid])
  mid++;
return mid;
}

void mergePrimes()
{
int i;
int j;

for (i=1;i<numThreads;i++)
{
  j=binSearch(aPrime[i],bPrime,1,numThreads-1);
  if (j<numThreads)
  {
    v[i+j-1].key=aPrime[i];
    v[i+j-1].index=i;
    v[i+j-1].sequenceName='a';
  }
  else
  {
    v[i+numThreads-1].key=aPrime[i];
    v[i+numThreads-1].index=i;
    v[i+numThreads-1].sequenceName='a';
  }

  j=binSearch(bPrime[i],aPrime,1,numThreads-1);
  if (j<numThreads)
  {
    v[i+j-1].key=bPrime[i];
    v[i+j-1].index=i;
    v[i+j-1].sequenceName='b';
  }
  else
  {
    v[i+numThreads-1].key=bPrime[i];
    v[i+numThreads-1].index=i;
    v[i+numThreads-1].sequenceName='b';
  }
}
}

void initQ()
{
int i;
int j;

for (i=2;i<=numThreads;i++)
{
  if (v[2*i-2].sequenceName=='a')
  {
    j=binSearch(v[2*i-2].key,b,0,arrSize-1);
    Q[i].aPos=v[2*i-2].index * arrSize/numThreads;
    Q[i].bPos=j;
  }
  else
  {
    j=binSearch(v[2*i-2].key,a,0,arrSize-1);
    Q[i].aPos=j;
    Q[i].bPos=v[2*i-2].index * arrSize/numThreads;
```

```
  }
}
}

void *seqMerge(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i;
int a1,b1,w;
int count;

i=threadWork->threadId+1;
a1=Q[i].aPos;
b1=Q[i].bPos;
w=a1+b1;
count=0;

if (i<numThreads)
{
  while (a1<Q[i+1].aPos && b1<Q[i+1].bPos)
  {
    if (a[a1]<=b[b1])
      c[w++]=a[a1++];
    else
      c[w++]=b[b1++];
    count++;
  }

  while (a1<Q[i+1].aPos)
  {
    c[w++]=a[a1++];
    count++;
  }
  while (b1<Q[i+1].bPos)
  {
    c[w++]=b[b1++];
    count++;
  }
}
else
{
  while (a1<arrSize && b1<arrSize)
  {
    if (a[a1]<=b[b1])
      c[w++]=a[a1++];
    else
      c[w++]=b[b1++];
    count++;
  }

  while (a1<arrSize)
  {
    c[w++]=a[a1++];
    count++;
  }
  while (b1<arrSize)
  {
    c[w++]=b[b1++];
    count++;
  }
}
printf("Thread %d wrote %d elements\n",threadWork->threadId,count);
barrier_wait(&barrier);
}

main()
{
thread_data_t *threadWork;
int seed;
int i,j,k,status;
pthread_t thread;

printf("enter number of threads\n");
scanf("%d",&numThreads);
```

```
printf("enter number of keys in each of the two tables\n");
scanf("%d",&arrSize);

a=(int*) malloc(sizeof(int)*arrSize);
if (!a)
{
  printf("malloc a failed\n");
  abort();
}
b=(int*) malloc(sizeof(int)*arrSize);
if (!b)
{
  printf("malloc b failed\n");
  abort();
}

arrSize2=2*arrSize;
c=(int*) malloc(sizeof(int)*arrSize2);
if (!c)
{
  printf("malloc c failed\n");
  abort();
}

v=(triple*) malloc((2*numThreads-1)*sizeof(triple));
if (!v)
{
  printf("malloc v failed\n");
  abort();
}

Q=(pair*) malloc((numThreads+1)*sizeof(pair));
if (!Q)
{
  printf("malloc Q failed\n");
  abort();
}

aPrime=(int*) malloc((numThreads+1)*sizeof(int));
if (!aPrime)
{
  printf("malloc aPrime failed\n");
  abort();
}

bPrime=(int*) malloc((numThreads+1)*sizeof(int));
if (!bPrime)
{
  printf("malloc bPrime failed\n");
  abort();
}

printf("enter seed\n");
scanf("%d",&seed);
srandom(seed);
j=k=0;
for (i=0;j+k<arrSize2;i++)
  if (j==arrSize)
    b[k++]=i;
  else if (k==arrSize)
    a[j++]=i;
  else
    switch (random() % 3) {
    case 0:
      a[j++]=i;
      break;
    case 1:
      b[k++]=i;
      break;
    case 2:
      a[j++]=i;
      b[k++]=i;
      break;
```

```
    }
  printf("input generated\n");

  #if DEBUG
  printf("a=\n");
  for (i=0;i<arrSize;i++)
  {
    printf("%5d",a[i]);
    if (i%NUMSPERLINES == NUMSPERLINES-1)
      printf("\n");
  }
  if (i%NUMSPERLINES != 0)
    printf("\n");

  printf("b=\n");
  for (i=0;i<arrSize;i++)
  {
    printf("%5d",b[i]);
    if (i%NUMSPERLINES == NUMSPERLINES-1)
      printf("\n");
  }
  if (i%NUMSPERLINES != 0)
    printf("\n");
  #endif

  primeInit();
  printf("prime arrays initialized\n");

  #if DEBUG
  printf("a'=\n");
  for (i=1;i<numThreads;i++)
    printf("%d %d\n",i,aPrime[i]);
  printf("b'=\n");
  for (i=1;i<numThreads;i++)
    printf("%d %d\n",i,bPrime[i]);
  #endif

  mergePrimes();
  printf("prime arrays merged\n");

  #if DEBUG
  printf("v=\n");
  for (i=2;i<=2*numThreads-2;i+=2)
    printf("%d %d %d %c\n",i,v[i].key,v[i].index,v[i].sequenceName);
  #endif

  Q[1].aPos=0;
  Q[1].bPos=0;
  initQ();
  printf("Q array initialized\n");

  #if DEBUG
  printf("Q=\n");
  for (i=1;i<=numThreads-1;i++)
    printf("%d %d %d\n",i,Q[i].aPos,Q[i].bPos);
  #endif

  printf("starting merges\n");

  barrier_init (&barrier, numThreads);
  for (i=1;i<numThreads;i++)
  {
    threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
    threadWork->threadId=i;
    status = pthread_create (
                  &thread, NULL, seqMerge, threadWork);
    if (status != 0)
    {
      printf("failed to create thread\n");
      exit(0);
    }
  }

  threadWork=(thread_data_t*) malloc(sizeof(thread_data_t));
```

```
threadWork->threadId=0;
seqMerge((void*) threadWork);

printf("merges complete\n");

#if DEBUG
printf("c=\n");
for (i=0;i<arrSize2;i++)
{
  printf("%5d",c[i]);
  if (i%NUMSPERLINES == NUMSPERLINES-1)
    printf("\n");
}
if (i%NUMSPERLINES != 0)
  printf("\n");
#endif

j=k=0;
for (i=0;i<arrSize2;i++)
  if (k==arrSize || j<arrSize && a[j]<b[k])
    if (c[i]==a[j])
      j++;
    else
    {
      printf("merge error 1 %d %d\n",i,j);
      abort();
    }
  else if (c[i]==b[k])
    k++;
  else
  {
    printf("merge error 2 %d %d\n",i,k);
    abort();
  }
printf("merge worked\n");
```

Problems for Notes 2:

1. Design a <u>function</u> for pthreads that uses an <u>arbitrary</u> number of threads to determine the sum of the elements of array C that have even-valued subscripts. Your function should be efficient and must use interleaving. Array C has exactly 1,000,000 elements. Code only the function, not main() or input/output.

2. The following C function is passed two shared arrays and a pointer to a shared integer for the result. Assuming that the threads have already been created, give pthreads code to make it execute efficiently in parallel. The results produced by the parallel version must be the same as the original sequential version.

```
minloc(doubles,ints,result)
double doubles[100000];
int ints[100000],*result;
{
int i,k;

k=0;
for (i=1; i<100000; i++)
   if (doubles[i] < doubles[k])
     k=i;
   else if (doubles[i] == doubles[k] && ints[i] < ints[k])
     k=i;
(*result)=k;
}
```

3. Give an efficient pthreads implementation of a concurrent function that corresponds to the following sequential function. Do not give the code to create the threads, i.e. just give the function. Your code must work with an arbitrary number of threads.

```
void f()
{
int i;

for (i=0; i<n; i++)
```

```
    a[i]=i*i;

  for (i=1; i<n; i++)
    b[i]=a[i-1] + a[i];
```

4. For what numbers of processors will the CREW merge be efficient?

5. Do a performance analysis (see Notes 1) comparing sequential and pthreads for the provided executions of shellsort and for exectutions using 5,000,000 keys.

Solutions:

1. This function is a variation of the program `smartMutexPT.c`:

```
void *sumEmUp(void *arg)
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i;
double localSum=0.0;
float startCPU;

startCPU=elapsedCPU();
for (i=2*threadWork->threadId;i<1000000;i+=2*numThreads)
{
  localSum+=C[i];
}
pthread_mutex_lock(&mutex);
sum+=localSum;
pthread_mutex_unlock(&mutex);
printf("Time used by thread %d is %f\n",threadWork->threadId,
  elapsedCPU()-startCPU);
barrier_wait(&barrier);
}
```

2. The key to this problem is to understand the meaning of the function <u>before</u> attempting to parallelize. If there is a single entry in `doubles` that contains the smallest value, then its subscript is returned. If multiple entries of `doubles` contain the smallest value, then the corresponding `ints` values are examined to determine the smallest value. The subscript of that value is then returned. If a tie on `ints` results, then the smallest subscript will be used. Before the parallel function is executed, `*result` has been intialized to `0`.

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;

minloc(threadId,doubles,ints,result)
int threadId
double doubles[100000];
int ints[100000],*result;
{
int i,k;

k=0;
for (i=1+threadId; i<100000; i+=numThreads)
  if (doubles[i] < doubles[k])
    k=i;
  else if (doubles[i] == doubles[k] && ints[i] < ints[k])
    k=i;
pthread_mutex_lock(&mutex);
if (doubles[k] < doubles[*result])
  (*result)=k;
else if (doubles[k] == doubles[*result] && ints[k] < ints[*result])
  (*result)=k;
else if (doubles[k] == doubles[*result] && ints[k] == ints[*result]
     && k<(*result))
  (*result)=k;
pthread_mutex_unlock(&mutex);
}
```

3. This problem is very straightforward and simply requires a barrier which was initialized earlier. It is also possible to coalesce the two loops into one loop to avoid some overhead.

```
void *f(void *arg)
```

```
{
thread_data_t *threadWork=(thread_data_t*) arg;
int i;

for (i=threadWork->threadId; i<n; i+=numThreads)
  a[i]=i*i;
barrier_wait(&barrier);
for (i=1+threadWork->threadId; i<n; i+=numThreads)
  b[i]=a[i-1] + a[i];
}
```

4. For what numbers of processors will the CREW merge be efficient?

   Step 1 (sampling) takes θ(1) time.

   Step 2 (merge samples via binary search) takes θ(lg P) time.

   Step 3 (determine remaining starting points via binary search) takes θ(lg (N/P)) time.

   Step 4 (merging) takes θ(N/P) time.

   Total time = θ(lg P + lg N + N/P)

   Work = θ(P lg P + P lg N + N)

   If P ≤ N/lg N, then work = θ(N)

5. Do a performance analysis ....

```
[bob@ketchup NOTES02]$ shellserial
enter number of keys
2000000
enter seed
123
CPU used is 16.650001
Elapsed time is 16.750000
[bob@ketchup NOTES02]$ shellPT
enter number of threads:2
enter number of keys
2000000
enter seed
123
CPU used by thread 0 is 9.620000
CPU used by thread 1 is 8.720000
Elapsed time for thread 1 is 9.750000
Elapsed time for thread 0 is 9.750000
Verifying sort . . .
sort succeeded
```

   Speed-up = 16.75/9.75 = 1.72
   Work = 2 * 9.75 = 19.5 (9.75 is the maximum elapsed time for the two threads)
   Efficiency = 16.75/19.5 = 0.86 (inefficiency caused by thread 1 waiting at barrier)

```
[bob@ketchup NOTES02]$ shellserial
enter number of keys
5000000
enter seed
765
CPU used is 50.420001
Elapsed time is 50.500000
[bob@ketchup NOTES02]$ shellPT
enter number of threads:2
enter number of keys
5000000
enter seed
765
CPU used by thread 0 is 28.160000
```

```
CPU used by thread 1 is 25.889999
Elapsed time for thread 1 is 28.250000
Elapsed time for thread 0 is 28.250000
Verifying sort . . .
sort succeeded
```

Speed-up = 50.5/28.25 = 1.79
Work = 2 * 28.25 = 56.5
Efficiency = 0.89