Principles of Concurrency and Parallelism

Lecture 6: Posix

2/7/12

Shared Memory

- Threads communicate by reading and writing to shared memory
- Easier transition from sequential programs
  - Don't have to construct new communication abstractions
- But, implicit communication via shared-memory raises complex issues of its own
  - Data races: concurrent (unintended) access to the same memory location
- How do we express concurrency and synchronization?
  - As language primitives (Java, C#, ...)
  - As library calls (Posix (Pthreads), Intel TBB)

Abstraction

- Shared Memory
  - Every thread can observe actions of other threads on non-thread-local data (e.g., heap)
  - Data visible to multiple threads must be protected (*synchronized*) to ensure the absence of *data races*
    - A data race consists of two concurrent accesses to the same shared data by two separate threads, at least one of which is a write
- Thread safety
  - Suppose a program creates *n* threads, each of which calls the same procedure found in some library
  - Suppose the library modifies some global (shared) data structure
  - Concurrent modifications to this structure may lead to data corruption

PThreads

- Exist within a process
  - But, independent control flow
  - share common process resources (like the heap and file descriptors)
    - changes made by one thread visible to others
    - pointers have meaning across threads
    - two threads can concurrently read and write to the same memory location
- Maintain their own
  - stack pointer
  - Registers
  - Pending and blocked signals
- Can be scheduled by the operating system

Structure

Programs can be decomposed into discrete (mostly) independent tasks

The points where they overlap should be easily discerned and amenable for protection

Three basic structures

master-worker *(agenda or blackboard)*
result-oriented *(dataflow)*
pipeline-oriented *(specialist)*

API

- Four major groups
  - Management (create, destroy, join, ...)
  - Mutexes (synchronization)
  - Condition variables (synchronization defined in terms of programmer-specified conditions)
  - Barriers
- include pthread.h header to gain access to Pthreads operations

Thread Creation

- Initially, your `main()` program comprises a single, default thread.
- `pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- arguments:
  - `thread:` An opaque, unique identifier for the new thread returned by the subroutine.
  - `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  - `start_routine`: the C routine that the thread will execute once it is created.
  - `arg:` A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

- The maximum number of threads that may be created by a process is implementation dependent.

Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int err;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      err = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (err){
         printf("ERROR; return code from pthread_create() is %d\n", err);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Passing Arguments

```
long *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (long *) malloc(sizeof(long));
   *taskids[t] = t;
   printf("Creating thread %ld\n", t);
   err = pthread_create(&threads[t], NULL, PrintHello,
                        (void *) taskids[t]);
   ...
}
```

Multiple Arguments

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   err = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
   ...
}
```

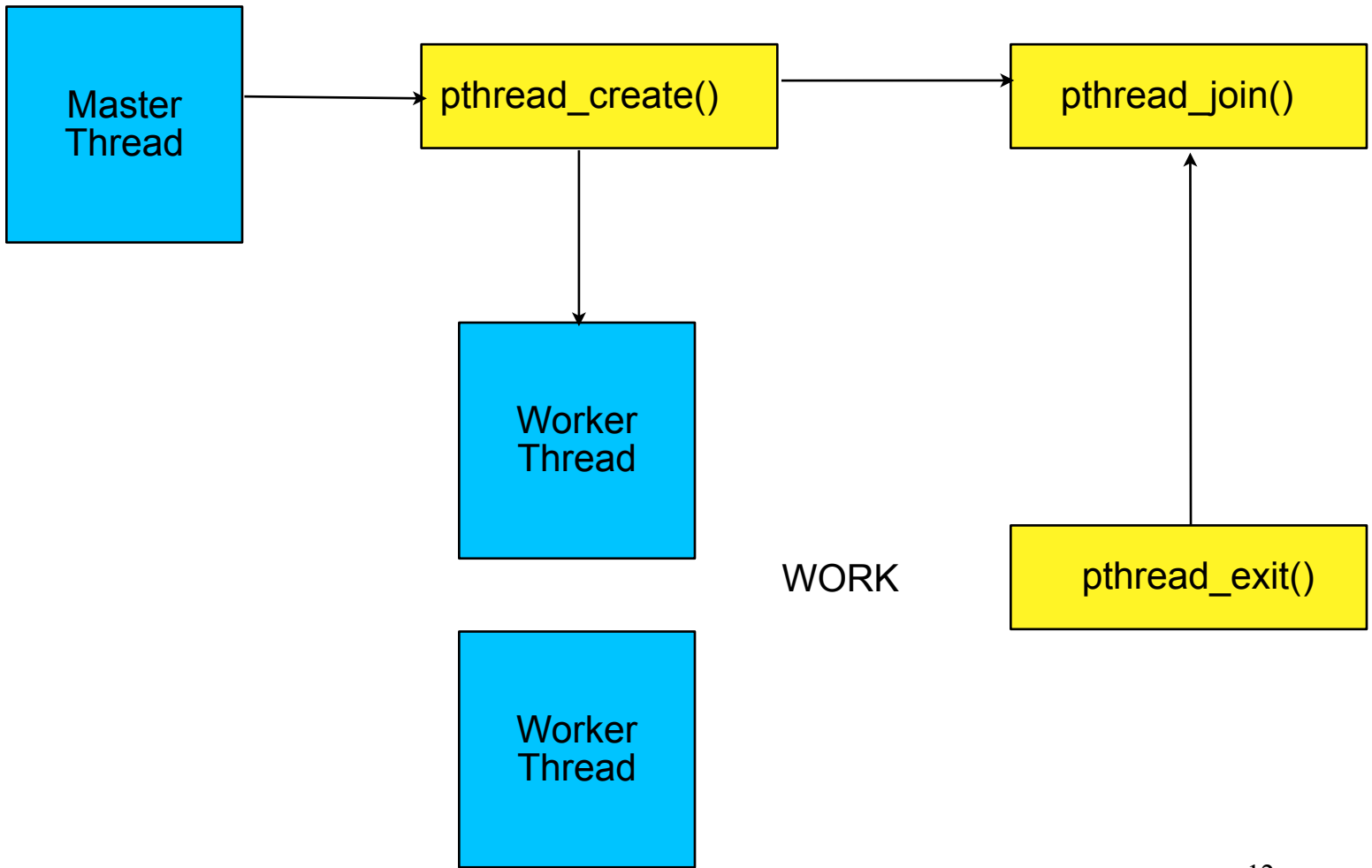
Example

What's wrong with the following?

```
int err;
long t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   err = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
   ...
}
```

Joining and Detaching Threads



Joining and Detaching

- `pthread_join()` blocks the calling thread until the specified `threadid` terminates
- A joining thread can match one `pthread_join()` call
- A thread created as detached can never be joined
- Use the attr argumentin a `pthread_create()` call to set joinable or detachable attributes

Example

```
#include <pthread.h>
...
#define NUM_THREADS   4

void *BusyWork(void *t) { ... pthread_exit((void*) t);   }

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   ...

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      err = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      ...         }
      }
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      err = pthread_join(thread[t], &status);
      ...
      printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Race Conditions

```
THREAD 1
a = data;
a++;
data += a;
```

```
THREAD 2
b = data;
b++;
data += b;
```

Assuming data = 0 initially, can data be 1 after the program completes?

Mutexes

- Protect access to shared data
- Methodology
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - One succeeds
  - Owner manipulates data protected by mutex
  - Owner unlocks
  - Another thread acquires the mutex, and repeats
  - Destroy the mutex

Mutexes

- Challenges:
  - make sure data is consistently protected by the same set of mutexes
  - make sure mutexes properly released
  - ensure deadlock-freedom
  - ensure progress (liveness)

Example

```
void *dotprod(void *arg)
{

   /* Define and use local variables for convenience */

   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   /*
   Perform the dot product and assign result
   to the appropriate variable in the structure.
   */

   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

   /*
     Lock a mutex prior to updating the value
     in the shared
     structure, and unlock it upon updating.
     */
     pthread_mutex_lock (&mutexsum);
     dotstr.sum += mysum;
     pthread_mutex_unlock (&mutexsum);

     pthread_exit((void*) 0);
}
```

Example (cont)

```
int main (int argc, char *argv[])
{
   long i;
   double *a, *b;
   void *status;
   pthread_attr_t attr;

   /* Assign storage and initialize values */
   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i<VECLEN*NUMTHRDS; i++)
    {
     a[i]=1.0;
     b[i]=a[i];
    }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;
```

Example (cont)

```
  pthread_mutex_init(&mutexsum, NULL);

  /* Create threads to perform the dotproduct  */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(i=0; i<NUMTHRDS; i++)
       {
    /*
    Each thread works on a different set of data.
    The offset is specified by 'i'. The size of
    the data for each thread is indicated by VECLEN.
    */
    pthread_create(&callThd[i], &attr, dotprod, (void *)i);
    }

    pthread_attr_destroy(&attr);

       /* Wait on the other threads */
    for(i=0; i<NUMTHRDS; i++)
       {
      pthread_join(callThd[i], &status);
    }

  /* After joining, print out the results and cleanup */
  printf ("Sum =  %f \n", dotstr.sum);
  free (a);
  free (b);
  pthread_mutex_destroy(&mutexsum);
  pthread_exit(NULL);
}
```

Condition Variables

- Allows threads to automatically synchronize based on the actual value of data
- Avoids the need for threads to continually poll to check if a condition is met

Example

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

Example (cont)

```
void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %ld, count = %d  Threshold reached.\n",
             my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
        my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some "work" so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}
```

Example (cont)

```
void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait
  routine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n", my_id);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

Example (cont)

```
int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```

Analysis

- Why use Pthreads?
  - portablity
  - performance
    - assuming optimized sequential code
    - and relatively little complex coordination
      - no inter-thread optimizations
- How do we quantify the effectiveness of a parallel program in terms of its sequential components?
  - speedup, overhead, decomposition

Amdahl's Law

$$\text{Speedup}_{enhanced}(f,S) = \frac{1}{(1-f)+\frac{f}{S}}$$

Here, *f* is the fraction of a computation that can be improved by a speedup *S*
When *f* is small, optimizations will have little effect

$$\text{Speedup}_{parallel}(f,n) = \frac{1}{(1-f)+\frac{f}{n}}$$

Here, *f* is the fraction of a sequential computation that can be improved by executing on *n* cores:

infinitely parallelizable (no scheduling overhead); remaining totally sequential

Assumes computation problem doesn't change with increase in cores; fraction of a program that is parallelizable remains fixed.

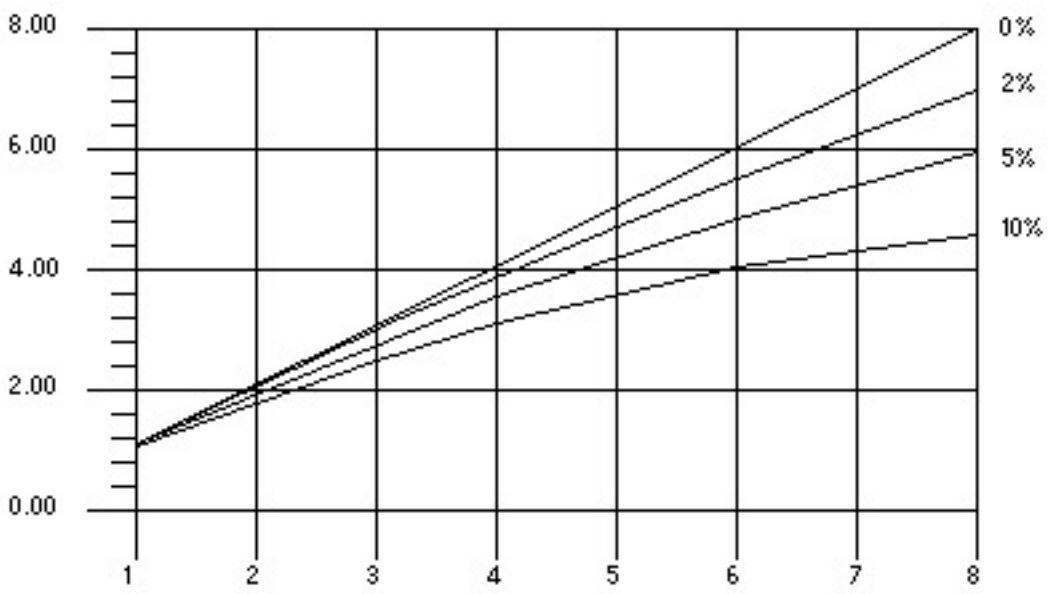
Example

- A program runs in 100 seconds on a machine; a multiply operation consumes 80% of this time.
  - How much do we need to improve the speed of the multiply operator to make the program run 4 times faster?
- A new processor is 20x faster on search queries than an existing one. Queries account for 70% of the time spent in a computation.
  - What is the speedup gained by using the new processor?

Example

- 90% of a calculation can be parallelized. What is the maximum speedup on 5 processors? 10 processors? 1000 processors?
- What if 99% of a calculation can be parallelized?

Speedup curves

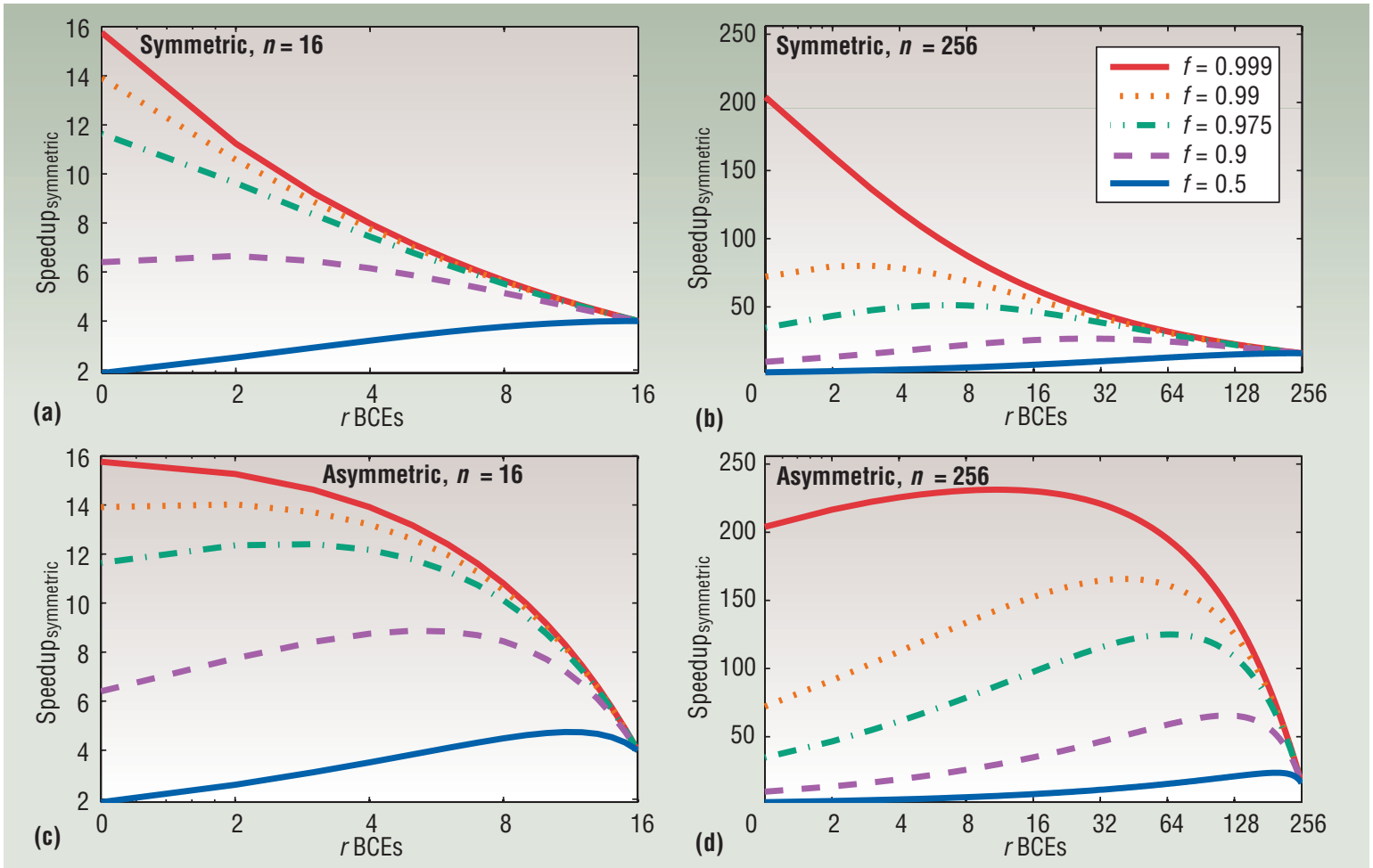


plotting sequential components
what happens when overheads are introduced?
how do we take problem size into account?

Corollaries

- What do the previous examples tell you about the main point of Amdahl's law?
  - performance of any system constrained by the speed of the slowest component
  - impact of performance improvement constrained by the parts of the program *not* targeted for improvement.

On Multicores (assume perf(r) = sqrt(r))



Implications

- Critical to increase fraction of computation that is parallelizable
- Using more tightly-coupled computation units per core can be beneficial
  - increasing individual core performance is essential (even if it increases the cost of each core)
  - denser chips soften the impact of Amdhal's law at scale
- Assymetric designs can lead to better speedups than symmetric designs

Readings

- Pthreads tutorial:
  - https://computing.llnl.gov/tutorials/pthreads/#PthreadsAPI
  - https://computing.llnl.gov/tutorials/parallel_comp/
- Monitors and condition variables
  - http://dl.acm.org/citation.cfm?id=361161
- Amdhal's Law and multicores
  - http://research.cs.wisc.edu/multifacet/papers/ieeecomputer08_amdahl_multicore.pdf