Scalable Algorithmic Techniques - focuses on data parallelism instead of task parallelism since task parallelism does scale well with # of processors, P (i.e., the number of tasks determines the parallelism -- however, each task can often have data parallelism)

General steps for designing parallel programs ("Foster's methodology"):
1. Partitioning: Divide computation and the data operated on into small tasks. Here the focus is identifying tasks that can be executed in parallel. (We might have true (RAW) data dependencies to consider.)
2. Communication: Determine what communication is needed to be carried out among the tasks identifed in step 1
3. Agglomeration or aggregation: Combine tasks and communications identified in the previous steps into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. Mapping: Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread get roughly the same amount of work

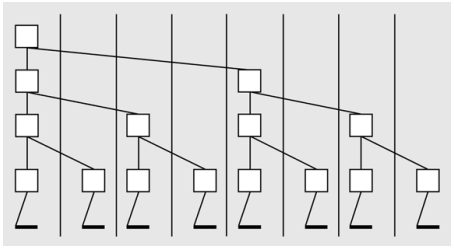
Guiding principle: *Parallel programs are more scalable when they emphasize blocks of computation -- typically the larger the block the better -- that minimize the inter-thread dependencies.*

Example: applies this guiding principle to array summation via tree-reduction where # processors P << n (array size).

Approach (1)	Approach (2)
• Create n/2 logically threads evenly across the P processors, so each processor had n/2P threads. • Have the logical threads perform the binary-tree reduction	• Have each processor sum its n/P local values • Have the P processors perform the binary reduction on a P local sums

a) Why is approach 2 better?

Approach (2) for tree-reduction, where:
- vertical lines denote threads,
- bold dark lines denote local computations,
- boxes denote partial sums in tree-reduction

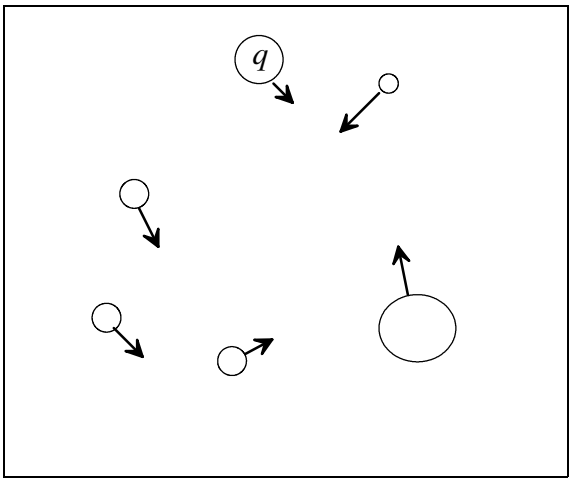


Dynamic Allocation of Data and Work: it might be hard or impossible to statically assign an even workload for several reasons:
- server processes client requests - requests determine the amount of work
- dynamic work created on the fly as the computation proceeds
- size of static data does not reflect amount of computation (i.e., hard to load balance)

Work Queue Solution - data structure for dynamically assigning work to threads or processes. A thread/process *producing* extra work puts it into the work queue. An idle thread/process remove (*consumes*) work from the queue to keep busy.

Types of work queues might depend on the problem being solved. Some options:
- FIFO queue - add new items to rear and remove items from the front
- LIFO stack - add new items to the top of the stack and remove items from the top (e.g., depth-first search)
- randomized queue - remove items randomly from queue
- priority queue - each item has a priority associated with it. Remove the item with highest priority (e.g, best-first search)

Chapter 6 has a couple larger "real-world" examples to demonstrate parallel program development. The first one is the n-body problem where we calculate the movement of n-bodies (e.g., n objects/planets in space, or n particles in a contain) over time. To be concrete, the book considers the motion of planets or stars in a 2D space.



a) What initial information would we need to know about each planet to calculate their motion over time?

b) How do planets "interact" to effect their motions?

Serial code:

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

The for each particle q:

Computer total force on q"

code needs to perform individual force calculations:

Nodes:

- row 0 are the forces on particle 0 by other particles, etc.
- matrix is "symmetric", except opposite forces are negated (two versions: *basic* and *reduced* utilitizing symmetry)

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

5. If we want to parallelize the calculation by applying the "guiding principle," then how might we map calculations to threads?

```
/* Function:  Thread_work
 * Purpose:   Execute individual thread's contribution
 *            to finding the positions and velocities
 *             of the particles.
 * In arg:    rank:   thread's rank (0, 1, . . . ,
thread_count-1)
 * Globals:   thread_count (in):  */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   int step;    /* Current step      */
   int part;    /* Current particle  */
   double t;    /* Current Time      */
   int first;   /* My first particle */
   int last;    /* My last particle  */
   int incr;    /* Loop increment    */

   Loop_schedule(my_rank, thread_count, n, BLOCK,
                 &first, &last, &incr);
   for (step = 1; step <= n_steps; step++) {
      t = step*delta_t;
      /* Particle n-1 will have all forces computed
       * after call to Compute_force(n-2, . . .) */
      for (part = first; part < last; part += incr)
         Compute_force(part);
      Barrier();
      for (part = first; part < last; part += incr)
         Update_part(part);
      Barrier();
#     ifndef NO_OUTPUT
      if (step % output_freq == 0 && my_rank == 0) {
         Output_state(t);
      }
#     endif
   }  /* for step */

   return NULL;
}  /* Thread_work */
```

```
/*--------------------------------------------------------------
 * Function:  Loop_sched
 * Purpose:   Return the parameters for a block or a cyclic
schedule
 *            for a for loop
 * In args:
 *    my_rank:       rank of calling thread
 *    thread_count:  number of threads
 *    n:             number of loop iterations
 *    sched:         schedule:  BLOCK or CYCLIC
 * Out args:
 *    first_p:       pointer to first loop index
 *    last_p:        pointer to value greater than last index
 *    incr_p:        loop increment
 */
void Loop_schedule(int my_rank, int thread_count, int n, int sched,
                   int* first_p, int* last_p, int* incr_p) {
   if (sched == CYCLIC) {
      *first_p = my_rank;
      *last_p = n;
      *incr_p = thread_count;
   } else {  /* sched == BLOCK */
      int quotient = n/thread_count;
      int remainder = n % thread_count;
      int my_iters;
      *incr_p = 1;
      if (my_rank < remainder) {
         my_iters = quotient + 1;
         *first_p = my_rank*my_iters;
      } else {
         my_iters = quotient;
         *first_p = my_rank*my_iters + remainder;
      }
      *last_p = *first_p + my_iters;
   }

}  /* Loop_schedule */
```

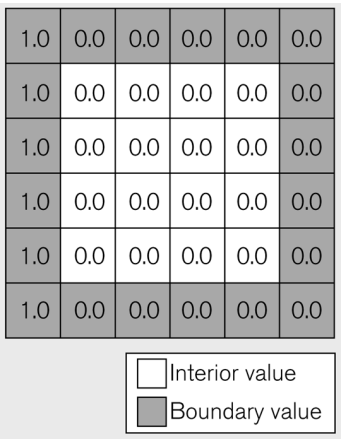
```
/* Function:  Compute_force
 * Purpose:   Compute the total force on particle part.  This
 *            version does *not* exploit the symmetry.
 * In arg:  part: the particle on which we're computing the total force
 * Global vars:
 *    curr (in):  current state of the system:  curr[i] stores the mass,
 *       position and velocity of the ith particle
 *    n (in):     number of particles
 *    forces (out): forces[i] stores the total force on the ith particle
*/
void Compute_force(int part) {
   int k;
   double mg;
   vect_t f_part_k;
   double len, len_3, fact;
   forces[part][X] = forces[part][Y] = 0.0;
   for (k = 0; k < n; k++) {
      if (k != part) {
         /* Compute force on part due to k */
         f_part_k[X] = curr[part].s[X] - curr[k].s[X];
         f_part_k[Y] = curr[part].s[Y] - curr[k].s[Y];
         len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
         len_3 = len*len*len;
         mg = -G*curr[part].m*curr[k].m;
         fact = mg/len_3;
         f_part_k[X] *= fact;
         f_part_k[Y] *= fact;

         /* Add force in to total forces */
         forces[part][X] += f_part_k[X];
         forces[part][Y] += f_part_k[Y];
      }
   }
}  /* Compute_force */
```

```
/* Function:  Update_part
 * Purpose:   Update the velocity and position for particle part
 * In arg:       part:    the particle we're updating
 * Global vars:  forces (in):   forces[i] stores the total force on
 *                               the ith particle
 *               n (in):        number of particles
 *               curr (in/out): curr[i] stores the mass, position and
 *                              velocity of the ith particle */
void Update_part(int part) {
   double fact = delta_t/curr[part].m;

   curr[part].s[X] += delta_t * curr[part].v[X];
   curr[part].s[Y] += delta_t * curr[part].v[Y];
   curr[part].v[X] += fact * forces[part][X];
   curr[part].v[Y] += fact * forces[part][Y];
}  /* Update_part */
```

```
#define DIM 2  /* Two-dimensional system */
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */

const int BLOCK = 0; /*Block partition of loop iterations*/
const int CYCLIC = 1; /*Cyclic partition of loop iter.s */

typedef double vect_t[DIM]; /*Vector type for position,etc*/

struct particle_s {
   double m;  /* Mass     */
   vect_t s;  /* Position */
   vect_t v;  /* Velocity */
};
```

HW #6: Successive Over-Relaxation (SOR) - often used in 3D form to solve differential equations such as Navier-Stokes equations for fluid flow.

6. 2D SOR - on each iteration replace all interior values by the average of their four nearest neighbors



a) Would it be best to perform static allocation or dynamic allocation (e.g., work queue) of the SOR calculations to threads?

b) How should we decompose the work among threads?

c) How do we synchronize the threads so all threads finish an iteration before any start the next iteration?

```c
/*  Programmer:  Mark Fienup
    File:        hw6.c
    Compiled by: gcc hw6.c -lpthread -lm
    Description:  2D over-relaxation program written using POSIX threads.
     Using barrier we implemented with condition variable and a  mutex.
*/
#include <math.h>
#include <stdio.h>
#include <limits.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>

#define MAXTHREADS 16/* max. # threads */

void * thread_main(void *);
void InitializeData();
void barrier();

pthread_mutex_t update_lock;
pthread_mutex_t barrier_lock;/* mutex for the barrier */
pthread_cond_t all_here;/* condition variable for barrier */
int count=0;/* counter for barrier */

int n, t, rowsPerThread;
double threshold;
double **val, **new;
double delta = 0.0;
double deltaNew = 0.0;

/* Command line args: matrix size, number of threads, threshold*/
int main(int argc, char * argv[])
{
    /* thread ids and attributes */
    pthread_t tid[MAXTHREADS];
    pthread_attr_t attr;
    long i, j;
    long startTime, endTime,seqTime,parTime;
    float myThreshold;

/* set global thread attributes */
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* initial mutex and condition variable */
    pthread_mutex_init(&update_lock, NULL);
    pthread_mutex_init(&barrier_lock, NULL);
    pthread_cond_init(&all_here, NULL);

    /* read command line arguments */
    if (argc != 4) {
        printf("usage: %s <matrix size> <number of threads> <threshold>\n",
               argv[0]);
        exit(1);
    } // end if

    sscanf(argv[1], "%d", &n);
    sscanf(argv[2], "%d", &t);
    sscanf(argv[3], "%f", &myThreshold);
    threshold = (double) myThreshold;

    rowsPerThread = n/t;
    InitializeData();
    printf("InitializeData done\n");
    for(i=0; i<t; i++) {
        pthread_create(&tid[i], &attr, thread_main, (void *) i);
    } // end for

    for (i=0; i < t; i++) {
      pthread_join(tid[i], NULL);
    } // end for

    printf("maximum difference:  %e\n", delta);

} // end main
```

```c
void InitializeData() {
    int i, j;

    new = (double **) malloc((n+2)*sizeof(double *));
    val = (double **) malloc((n+2)*sizeof(double *));

    for (i = 0; i < n+2; i++) {
        new[i] = (double *) malloc((n+2)*sizeof(double));
        val[i] = (double *) malloc((n+2)*sizeof(double));
    } // end for i

  /* initialize to 0.0 except to 1.0 along the left boundary */
    for (i = 0; i < n+2; i++) {
        val[i][0] = 1.0;
        new[i][0] = 1.0;
    } // end for i
    for (i = 0; i < n+2; i++) {
        for (j = 1; j < n+2; j++) {
            val[i][j] = 0.0;
            new[i][j] = 0.0;
        } // end for j
    } // end for i
} // end InitializeData

void barrier(long id) {
    pthread_mutex_lock(&barrier_lock);
    count++;
    printf("count %d, id %d\n", count, id);
    if (count == t) {
        count = 0;
        pthread_cond_broadcast(&all_here);
    } else {
        while(count < t) {
           while(pthread_cond_wait(&all_here, &barrier_lock)!=0);
        } // end while
    } // end if
    pthread_mutex_unlock(&barrier_lock);
} // end barrier
```

Handling "Hard" Problems: For many optimization problems (e.g., *NP-Complete* problems: TSP, knapsack, job-scheduling), the best known algorithms have run-time's that grow exponentially, $O(2^n)$. Thus, you could wait centuries for the solution of all but the smallest problems!

Ways to handle these "hard" problems:

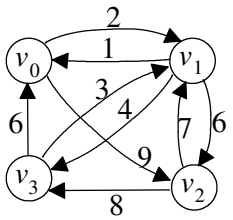
- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the 2^n worse solutions, e.g, Backtracking and Branch-and-Bound
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., $O(n^3)$) solution. e.g., Fractional Knapsack problem, TSP problem satisfying the triangle inequality
- Use an approximation algorithm to find a good, but not necessarily optimal solution

Backtracking general idea:

- Search the "*state-space tree*" using depth-first search to find a suboptimal solution quickly
- Use the best solution found so far to prune partial solutions that are not "promising,", i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time. However, in the worst case, the algorithm is still exponential.

7. Ch 6: *Traveling Salesperson Problem* (TSP) -- Find an optimal (i.e., minimum length) tour when at least one tour exists. A *tour* (or *Hamiltonian circuit*) is a path from a vertex back to itself that passes through each of the other vertices exactly once. (Since a tour visits every vertice, it does not matter where you start, so we will generally start at v_0.) What are the length of the following tours?



a) $[v_0, v_1, v_2, v_3, v_0]$

b) $[v_0, v_2, v_1, v_3, v_0]$

c) List another tour starting at v_0 and its length.

d) For a graph with "n" vertices ($v_0, v_1, v_2, \ldots, v_{n-1}$), one possible approach to solving TSP would be to brute-force generate all possible tours to find the minimum length tour. "Complete" the following decision tree to determine the number of possible tours.