Problem 1 (10 pts): Suppose you are given a program that does a fixed amount of work, and some fraction $s$ of that work must be done sequentially. The remaining portion of the work is perfectly parallelizable on $P$ processors. Assuming $T_1$ is the time taken on one processor, derive formula for $T_p$, the time taken on $P$ processors. Use this to get a formula giving an upper bound on the potential speedup on $P$ processors. Explain why it is an upper bound.

Answer: $S_p \leq \frac{1}{s + \frac{s-1}{p}}$

The sequential work takes as long as on one processor. If the remainder is perfectly parallelized, it will be accelerated by a factor of $p$. If there is redundant work performed, communication, or synchronization, this factor may be reduced.

Problem 2 (10 pts): Describe an example where a good parallel algorithm must be based on a serial algorithm that is different from the best serial algorithm since the later cannot be parallelized.

Answer: The sequential version of the grid solver problem cannot be easily parallelized. Therefore, application knowledge is needed to design a different algorithm (red-black ordering) for parallelization.

Problem 3 (10 pts): In the shared address space parallel equation solver, why do we need all the three barriers in a while loop iteration? What problems will we have if we eliminate them?

Answer: The barrier in line 16a ensures that all processes have finished their initialization of the global diff variable before any process starts performing the next iteration. The reason is that otherwise some processes may perform their work for the next iteration, including accumulating their private mydiff variables into the
shared diff, but after that another process that was slowed down somehow may set the shared diff to 0 at the beginning of its iteration, thus wiping out the accumulations of the mydiffs of other processes that had proceeded faster. This zeroing out of the diff variable may cause the computation to appear to have converged when the processes perform the check in line 25e after the barrier, even though it in fact hasn’t.

The barrier in line 25d ensures that all processes reach here before they check if done and ensure that all processes get the same answer. If the barrier is removed, processes that finish earlier may find done=1 because other processes have not accumulated their mydiff to diff. The early processes would exit prematurely.

The barrier in line 25f ensures that no process initializes the shared diff variable to be zero for the next iteration before all have tested its value for convergence in the current iteration and reached a consistent verdict. Otherwise, some processes may think the computation has not converged and will enter the next iteration, one of these may zero out the diff variable, and then some processes that have not yet performed the test from the previous iteration may read the zero or low value for diff, think the computation has converged, and not enter the next iteration. The processes that entered the next iteration will then hang waiting for others to arrive at the first barrier in that iteration (i.e. the barrier in line 16a), but those others will never arrive.

**Problem 4 (10 pts):** What are the advantages and disadvantages of using distributed task queues (as opposed to a global task queue) to implement load balancing? Do small tasks inherently increase communication, contention, and task management overhead in each case?

**Answer:** There are two major advantages to using distributed task queues. One is an increase in locality. If orchestrated properly, a process may obtain tasks from its own queue most of the time, and this queue may be in its local memory. The second is a reduction in contention. With a single queue, all processes access and contend for a single queue and for the locks on that queue. This is a major problem when tasks are small and accesses to the queue are frequent.

The major potential disadvantages of distributed queues have to do with complexity. First, the complexity of distributing work among queues: how does a process decide which queue to put new tasks in (the usual practice is to put tasks in its own queue). Second, complexity due to task stealing policies: which queues to steal from, how much to steal, etc. Third, termination detection: when stealing tasks from multiple queues into which new tasks may be added dynamically, how does a process determine when all the work is done and it should not try to steal any more? Finally, in some cases if the processor on which the process managing a centralized queue runs does little more than service queue requests, i.e. does not
participate much in the main computation, then it can be better to use a centralized queue on that processor (as long as there is enough bandwidth for servicing queue requests) than to distribute the queue and perhaps affect other busy processors. In general, distributed queues are used unless tasks are very big and accesses to a centralized queue infrequent.

For the second part of the question, even assuming that tasks are accessed one at a time from the queues, small tasks do not inherently increase communication and contention much; these depend on how well locality is achieved in queue management. For example, if a process is accessing its own queue almost all the time, then small tasks will not increase communication and contention much. However, small tasks will increase the overhead of queue management, unless we dequeue and enqueue multiple tasks at a time.

**Problem 5 (10 pts):** What are the advantages and disadvantages of static task assignment and dynamic task assignment? In what scenarios is dynamic assignment more desirable than static assignment?

Answer:

Static task assignment
Advantages: low runtime overhead, predictable performance and better data locality.
Disadvantage: possible load imbalance

Dynamic task assignment
Advantage: better load balance
Disadvantage: high task management overhead, excessive communications, and poor data locality.

**Problem 6 (20 pts):** Using data parallel programming mode to write a parallel program that adds two vectors (Hint: use the DECOMP and for_all statements to write the pseudo code):

```c
float v1[100], v2[100];
int nprocs; /* number of processes */
for (i = 0; i < 100; i++)
  v1[i] = v1[i] + v2[i];
```

Answer:

```c
float v1[100], v2[100];
int nprocs; /* number of processes */

int main()
begin
```
read(nprocs);
initialize(v1);
initialize(v2);
Add(v1, v2);
end

procedure Add(v1, v2)
begin
DECOMP v1[BLOCK, nprocs];
DECOMP v2[BLOCK, nprocs];
for_all (i = 0; i < 100; i++)
  v1[i] = v1[i] + v2[i];
end_for_all
end

Problem 7 (30 pts): Using shared memory model and message passing model to write parallel programs that add up the elements of a float-point array (A), whose size (n) and data are input by users. ‘nprocs’ processes are used. You can assume that n mod nprocs = 0. (pseudo code only)

float *A;
int n; /* size of A */
int nprocs; /* number of processes */
float total = 0;

for (i = 0; i < n; i++)
  total += A[i];

Answer:

Based on shared memory model:

int n, nprocs; /* matrix dimension and number of processors to be used */
float *A; /* A is a global, shared array */
float total = 0;
LOCKDEC(total_lock); /* declaration of lock to enforce mutual exclusion */

int main()
begin
  read(n);
  read(nprocs);
  A ← G_MALLOC (a n float array);
  initialize(A); /* initialize A */
  CREATE (nprocs–1, Add, A);
  Add(A); /* main process becomes a worker too*/


```
WAIT_FOR_END(nprocs-1); /*wait for all child processes created to terminate*/
end main

procedure Add(A)
  float *A; /*A is the shared array*/
  int i;
  int mymin, mymax;
  float myTotal = 0;
  begin
    mymin = (pid * n/nprocs); /* assume that n is exactly divisible by nprocs */
    mymax = mymin + n/nprocs – 1;
    BARRIER(bar, nprocs);
    for i ← mymin to mymax do /* for each of my rows */
      mytotal += A[i];
    endfor
    LOCK(total_lock); /* update global total */
    total += myTotal; /* critical section */
    UNLOCK(total_lock);
  end procedure

Based on message passing model:
int n, nprocs; /* array size and number of processors to be used*/
float *myA;
float mytotal = 0
int main()
begin
  read(n);
  read(nprocs);
  CREATE(nprocs-1, Add);
  Add(); /*main process becomes a worker too*/
  WAIT_FOR_END(nprocs-1); /*wait for all child processes created to terminate*/
end main

procedure Add()
begin
  int i, pid, n’ = n/nprocs;
  int temptotal = 0;
  myA ← malloc(n’ * sizeof(float)); /*my assigned elements of A*/
  initialize(myA);

  for i ← 0 to n’ do /* for each of my elements */
```
mytotal += myA[i];

if (pid != 0) then
    SEND(mytotal, sizeof(float), 0, TOTAL);
else
    /* pid 0 does this */
    for i ← 1 to nprocs-1 do /* process 0 holds global total */
        RECEIVE(&temptotal, sizeof(float), *, TOTAL);
        mytotal += temptotal; /* accumulate into
    mytotal of pid0 */
    endif
endif
end procedure