Problem 1. Greedy scheduler

Recall the greedy scheduler we studied in class and recall that we proved that the running time of a program with work $W$ and span $S$ on $P$ processors is upper bounded by $T_P \leq W/P + S$. Now we are asking you to analyze a slightly broken scheduler which operates as follows:

- On odd time steps (time steps 1, 3, 5, ...) the scheduler operates like the greedy scheduler. That is, if there are greater than or equal to $P$ ready nodes, it arbitrarily picks $P$ of them and executes them on the $P$ processors. If there are fewer than $P$ ready nodes, then it executes all the ready nodes.
- On even time steps, $P - 1$ processors goof off and only one processor does any work. It arbitrarily picks one ready node and executes it.

Provide an upper bound on the execution of a program with work $W$ and span $S$ using this scheduler and argue for the correctness of this bound. You do not have to repeat the greedy scheduler proof. If necessary, you can simply cite the steps of the proof or the result itself.

Problem 2. Fixing races

```
cilk_for i ← 1 to n
do x ← x + A[i]
```

The code above has a race on variable $x$. Rewrite the code so that it is free of determinacy races but without using a reducer.

Problem 3. Work and span analysis

Recall the matrix multiplication code without using a temporary matrix we covered in lecture 3. In the code shown in lecture, we recur down to base case of $1 \times 1$ matrix. Often time, to get good performance in parallel code — especially when the application has ample parallelism, it’s a good idea to coarsen the base case. Thus instead of base case of size $1 \times 1$, we coarsen the recursion so that we stop at base case of matrix sized $k \times k$, and simply do a serial triple nested loops to compute the matrix multiplication in the base case. With this change, what are the resulting work, span, and parallelism of the matrix multiplication code?
Problem 4. Cilk’s work-stealing scheduler

In the analysis of Cilk’s work-stealing scheduler, we are able to show the time bound $T_P \leq T_P + O(T_\infty + \lg(\frac{1}{\epsilon}))$ by showing that a critical strand in the augmented dag $G'$ is always either being executed or sitting on top of some worker’s deque. How would the time bound change if say, a critical strand in the augmented dag $G'$ is always either being executed or sitting in the top-two slots in some worker’s deque? Explain your answer.

Problem 5. Cilk’s work-stealing scheduler

In lecture 8, we talked about the lack of serial-parallel reciprocity (SP-reciprocity) in Cilk-5’s implementation. We also mentioned that, Cilk Plus took the design choice that sacrifices the space bound ($S_P \leq PS_1$) in order to regain SP-reciprocity. In Cilk Plus, whenever a worker needs to work steal, it simply gets another new stack and pushes new invocations associated with the newly stolen work onto the new stack. In this strategy, say we still want to ensure that the parallel execution does not use too many stacks, so we have the runtime system maintains a stack pool that contains $kP$ number of stacks (where $k$ is a user defined constant). Whenever a worker needs a new stack, it gets the stack from this stack pool. Whenever all functions on a stack finishes, the stack gets returned back to the stack pool. If a worker needs a new stack and the stack pool is empty, however, we simply have the worker spin-wait until some stack gets returned. Is this a good strategy? If so, why? If not, why not?

Problem 6. Race detection in Cilk

In the SP-Bags algorithm, we assume that a Cilk procedure is always spawned and that one cannot spawn an ordinary C function. How would you change the algorithm if we are race detecting a Cilk Plus computation, where a Cilk function can be called, and one can spawn an ordinary C functions? Justify your answer.