Problem 1: (50 points) While we understand that the fan-in-tree structured communication pattern can alleviate ‘hot spot’ contention to improve performance, can you write a parallel program based on the message passing model to sum \( n \) integers up using \( p \) processors.

Solution:

```c
int n, p;    /* array size and number of processors to be used*/
int *myA;
int total = 0
int main()
begin
  read(n);
  read(p);
  CREATE (p-1, Add);
  Add();      /*main process becomes a worker too*/
  WAIT_FOR_END (p–1);  /*wait for all child processes created to terminate*/
end main

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

  for i ← 0 to mysize do   /* for each of my elements */
    mytotal += A[i];      /* get local total */

  if (2*pid+1 <= p-1) { /* I have left child */
    RECEIVE(&temptotal, sizeof(int), 2*pid+1, TOTAL);
  }
end Add
```

Flat Tree structured

Contention

Little contention

Tree structured
mytotal += temptotal; /* accumulate the sum from left child into mytotal */
if (2*pid+2 <= p-1) { /* I have right child */
    RECEIVE(&temptotal, sizeof(int), 2*pid+2, TOTAL);
    mytotal += temptotal; /* accumulate the sum from right child into mytotal */
}
if (pid != 0)
    SEND(mytotal, sizeof(int), (pid-1)/2, TOTAL);

Problem 2: (50 points) You have been given the job of creating a word count program for a major book publisher. After processing all the words in a book, you should produce the number of distinct words used in the book as well as what they are. You will be working on a shared memory multiprocessor (SMP) with 32 processors. Your only stated interface is get_words(char *word[]), which takes as its parameter an array of character strings (words) and on return places in the array the book’s next 1000 words to be counted. The main work each processor will do should look like this:

while (get_words(word)) {
    for (i = 0; i < 1000; i++) {
        /* list is a string array storing all the distinct words that have been encountered so far.
         * You can sequentially search the list to see if there is a match
         */
        if word[i] is not in list {
            add word[i] to list;
            increment the count;
        }
    }
}

Write a parallel program based on shared memory model to complete the work. You can assume that there are 32 processes, each mapping to a processor in a one-to-one fashion. While the problem allows you a lot of flexibility to write a correct program, you must attempt to write an efficient one that minimizes space and synchronization overhead. For example, a process should not hold a mutual exclusion lock during the entire period when it searches the list to check if a word is in it. In addition to write the program, you should describe your design for improving efficiency and why it works.

Solution:

char list[MAX_DISTINCT_WORDS][MAX_WORD_SIZE];
int cur_list_size = 0;
int nprocs = 32; /* number of processes */
LOCKDEC(list_lock); /* declaration of lock to enforce mutual exclusion */
int main()
begin
    CREATE(nprocs-1, words_count);
    words_count();
    WAIT_FOR_END (nprocs–1);  /*wait for all child processes created to
terminate*/
end main

procedure words_count()
{
    char words[1000][MAX_WORD_SIZE];
    int num; /* number of words fetched */
    int my_list_size;
    int i, j;
    int found;

    while (num = get_words(word)) {
        for (i = 0; i < num; i++) {
            my_list_size = cur_list_size;

            found = 0;
            for (j = 0; j < my_list_size; j++)
                if (strcmp(word[i], list[j]) == 0) { /* if a match is found */
                    found = 1;
                    break;
                }

        }

        if (found == 0) {
            LOCK(list_lock);
            for (j = my_list_size; j < cur_list_size; j++)
                if (strcmp(word[i], list[j]) == 0) { /* if a match is found */
                    break;
                }
            if (found = 0) {
/* new word is always at the list tail */
                strcpy(list[cur_list_size], word[i]);
                cur_list_size++;
            }
            UNLOCK(list_lock);
        }
    }
}