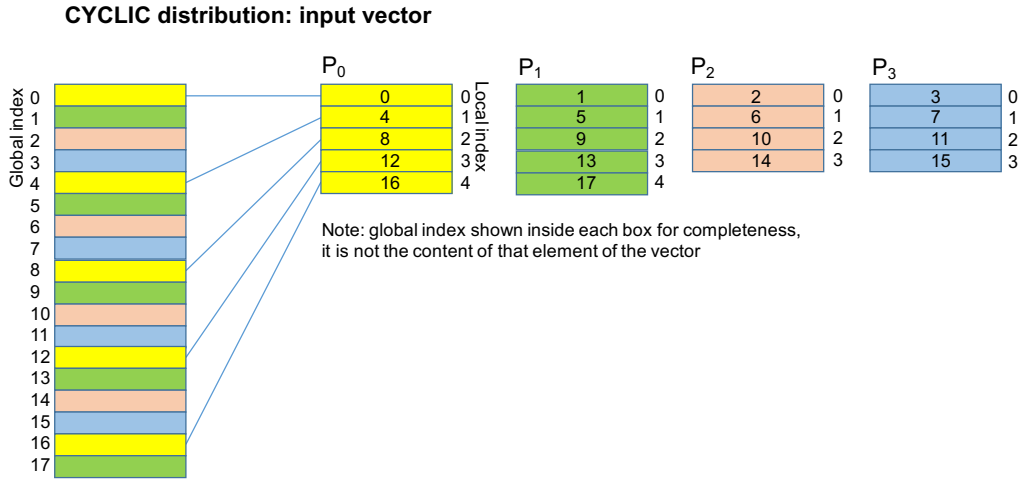
(Pseudo-)Problema 6 (ampliado para considerar memoria distribuida)

a1) Función `FindBounds`, data decomposition CYCLIC para vector `input`. Arquitectura memoria compartida.




```
void FindBounds(int * input, int size, int * min, int * max) {
int tmin=*min, tmax=*max; // reducción no permitida sobre punteros

#pragma omp parallel reduction(max: tmax) reduction(min: tmin)
      {
      int i_start = omp_get_thread_num();
      int howmany = omp_get_num_threads();
      for (int i=i_start; i<size; i += howmany) {
            if (input[i]>(tmax)) (tmax)=input[i];
            if (input[i]<(tmin)) (tmin)=input[i];
            }
      }
*min=tmin; *max=tmax;
}
```

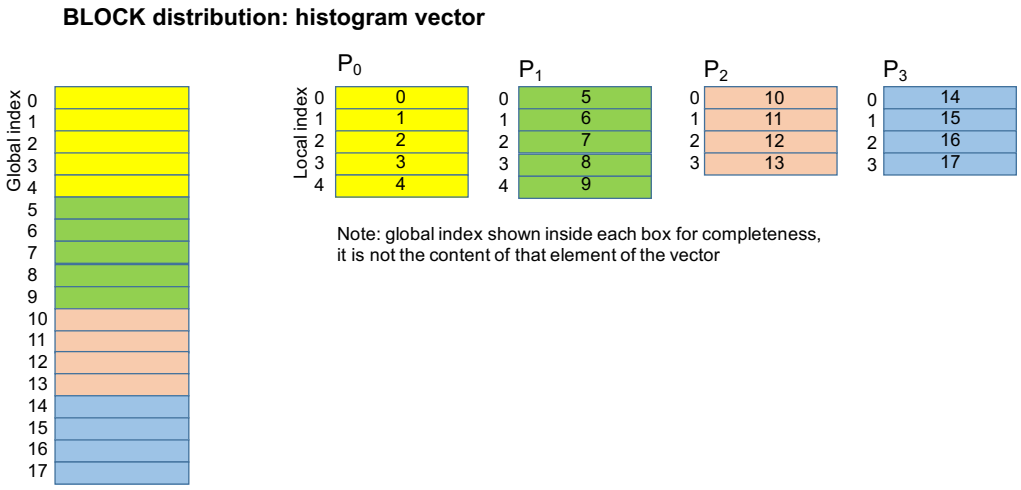
a2) Función `FindBounds`, data decomposition CYCLIC para vector `input`. Arquitectura memoria distribuida. Versión OpenMP (no valido como código paralelo, sólo para entender como cambia la indexación de las estructuras de datos).

```
void FindBounds(int * input, int size, int * min, int * max) {
int tmin=*min, tmax=*max; // reducción no permitida sobre punteros

#pragma omp parallel reduction(max: tmax) reduction(min: tmin)
      {
      int who = omp_get_thread_num();
      int howmany = omp_get_num_threads();
      int rem = size % howmany;
      int i_end = (size / howmany) + (who < rem ? 1 : 0);
      for (int i=0; i<i_end; i++) {
            if (input[i]>(tmax)) (tmax)=input[i];
            if (input[i]<(tmin)) (tmin)=input[i];
            }
      }
*min=tmin; *max=tmax;
}
```

a3) Función `FindBounds`, data decomposition CYCLIC para vector `input`. Arquitectura memoria distribuida. Versión MPI (no incluye operación de reducción, ver apartado c)).

```
void FindBounds(int * input, int size, int * min, int * max) {
int who, howmany;

MPI_Comm_rank(MPI_COMM_WORLD, &who);
MPI_Comm_size(MPI_COMM_WORLD, &howmany);
int rem = size % howmany;
int i_end = (size / howmany) + (who < rem ? 1 : 0);
for (int i=0; i<i_end; i++) {
      if (input[i]>(*max)) (*max)=input[i];
      if (input[i]<(*min)) (*min)=input[i];
       }
}
```

b1) Función `FindFrequency`, data decomposition BLOCK para vector `histogram`. Arquitectura memoria compartida




```
void FindFrequency(int * input, int size ,
                   int * histogram, int min, int max) {
#pragma omp parallel private(tmp)
    {
    int who = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int i_start = who * (HIST_SIZE/howmany) +
              (who < (HIST_SIZE%howmany) ? who : (HIST_SIZE%howmany));
    int i_end = i_start + (HIST_SIZE/howmany) +
              (who<(HIST_SIZE%howmany));
    for (int i=0; i<size; i++) {
        int tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
        if ((tmp>=i_start) && (tmp<i_end))
            histogram[tmp]++;
        }
    }
```

b2) Función `FindFrequency`, data decomposition BLOCK para vector `histogram`. Vector input replicado. Arquitectura memoria distribuida. Versión OpenMP (no valido como código paralelo, sólo para entender como cambia la indexación de las estructuras de datos).

```
void FindFrequency(int * input, int size ,
                   int * histogram, int min, int max) {

#pragma omp parallel
    {
    int who = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int i_start = who * (HIST_SIZE/howmany) +
              (who < (HIST_SIZE%howmany) ? who : (HIST_SIZE%howmany));
    int i_end = i_start + (HIST_SIZE/howmany) +
              (who<(HIST_SIZE%howmany));
    for (int i=0; i<size; i++) {
        int tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
        if ((tmp>=i_start) && (tmp<i_end))
            histogram[tmp-i_start]++;
        }
    }
}
```

b3) Función `FindFrequency`, data decomposition BLOCK para vector `histogram`. Vector input replicado. Arquitectura memoria distribuida. Versión MPI (no incluye operación colectiva de gather, ver apartado c)).

```
void FindFrequency(int * input, int size ,
                   int * histogram, int min, int max) {
int who, howmany;

MPI_Comm_rank(MPI_COMM_WORLD, &who);
MPI_Comm_size(MPI_COMM_WORLD, &howmany);

int i_start = who * (HIST_SIZE/howmany) +
              (who < (HIST_SIZE%howmany) ? who : (HIST_SIZE%howmany));
int i_end   = i_start + (HIST_SIZE/howmany) +
              (who<(HIST_SIZE%howmany));
for (int i=0; i<size; i++) {
      int tmp = (input[i] - min) * HIST_SIZE / (max - min - 1);
      if ((tmp>=i_start) && (tmp<i_end))
            histogram[tmp-i_start]++;
      }
}
```

c) Para `frequency` sería necesario que P_0 distribuyera el vector entre todos los procesadores (comunicación colectiva tipo “scatter”). Dado que no se inicializa, también sería valido si no se realiza esta comunicación. Sin embargo, es necesario que P_0 recoja del resto de procesadores la porción del vector frequency calculado (colectiva tipo “gather”) después de la ejecución de `FindFrequency`.

Para la variable `max` es necesario que P_0 la replique en el resto de procesadores (comunicación colectiva tipo “broadcast”) antes de la ejecución de `FindBounds`

(dado que está inicializada). Después de la ejecución de `FindBounds` P_0 deberá combinar los máximos parciales en cada procesador (comunicación colectiva tipo “reduce”) y volver a hacer un “broadcast” antes de iniciar la ejecución de `FindFrequency` con el objetivo de distribuir a todos los procesadores el valor máximo encontrando. Esta operación colectiva “reduce-broadcast” conjunta también existe en MPI y se denomina “allreduce”.