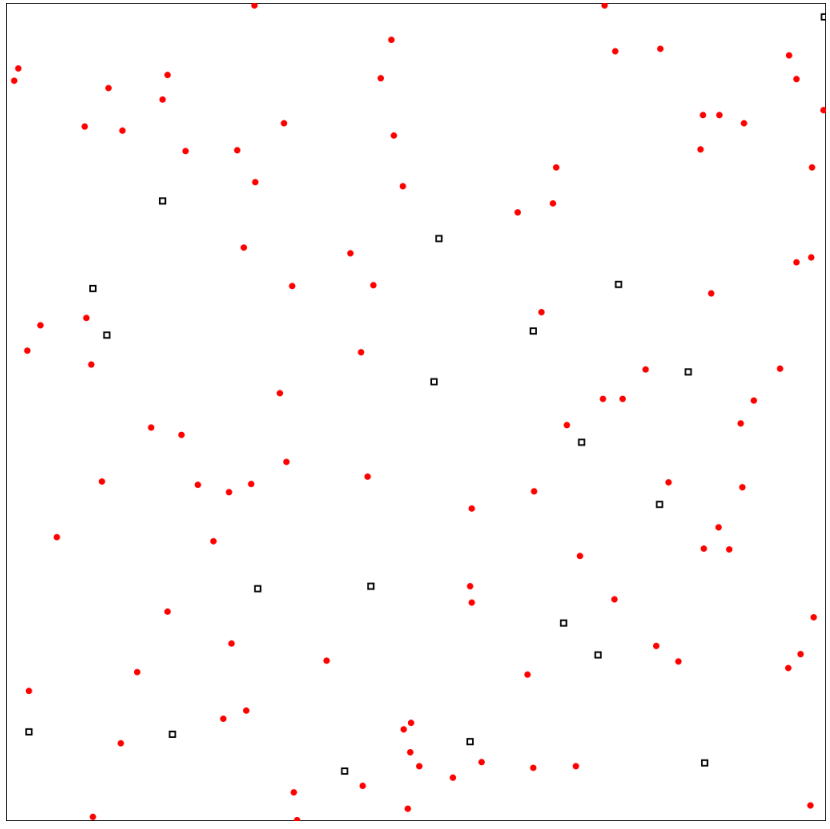
Paralelización de Disperse Construction

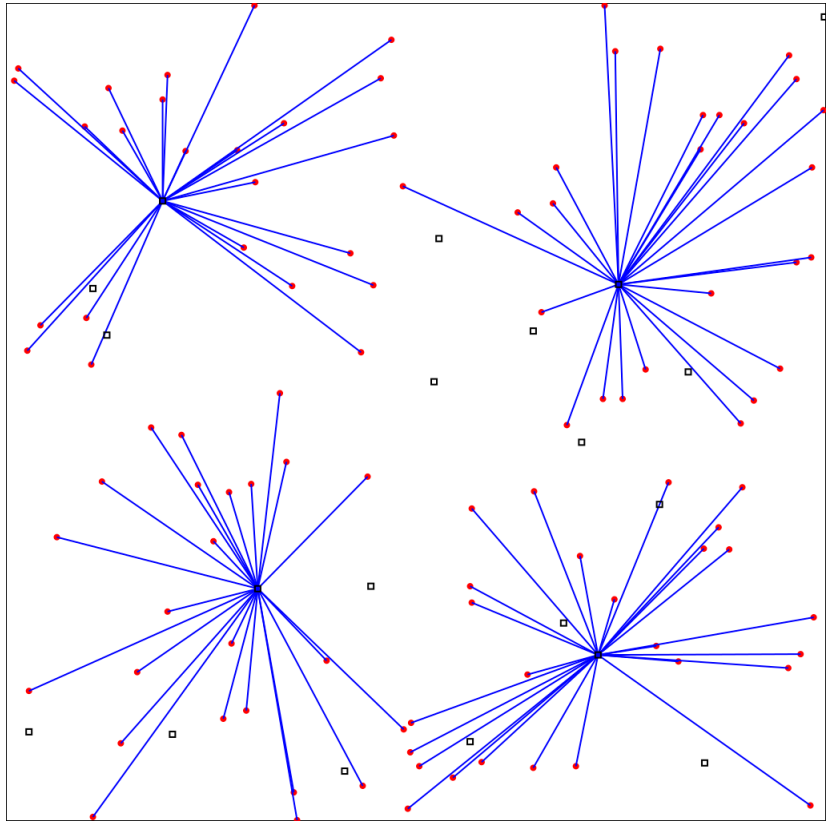
Programación Paralela Aplicada

Francisco Casas B. (@autopawn)

Problema



Problema

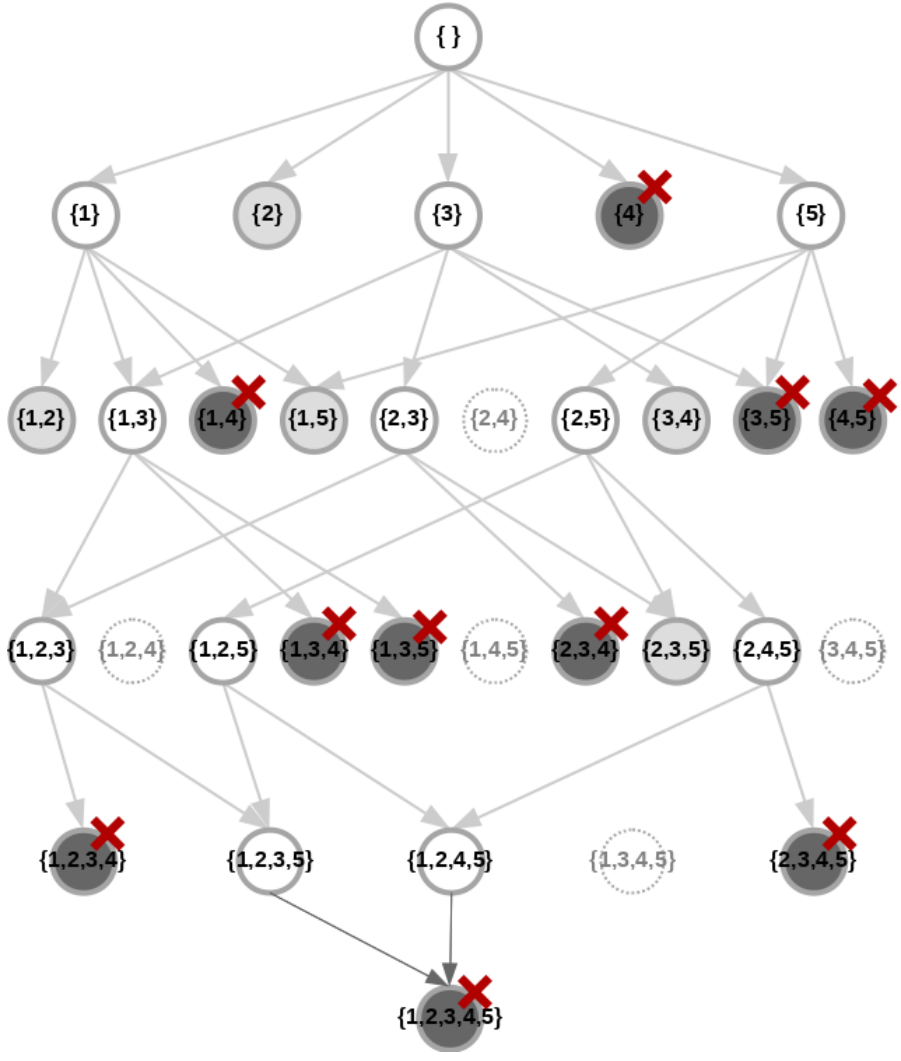


Algorimo

Algorithm 1 Disperse Construction

procedure DISPERSECONSTRUCTION($Z, \texttt{PZ}, \texttt{MD} \mid \Phi, d$)
  $P_0 \leftarrow \{\phi\}$
  for $n = 0$ to $\texttt{MD}$ **do**
    $B_{n+1} \leftarrow$EXPAND($P_n, Z \mid \Phi$)
    $P_{n+1} \leftarrow$ REDUCE($B_{n+1}, \texttt{PZ} \mid \Phi, d$)
  end for
  $\hat{P} \leftarrow \bigcup_{n=0}^{\texttt{MD}} P_n$
  $R \leftarrow \{$LOCALSEARCH$(p, Z \mid \Phi) \quad | \quad p \in \hat{P}\}$
  return R sorted by decreasing value on Φ.
end procedure

Descripción del algorimo



Proceso de expansión

Algorithm 2 Expansion process

procedure EXPAND($P, Z \mid \Phi$)
  $B \leftarrow \phi$
  for $p \in P$ **do**
    for $z \in (Z \setminus p)$ **do**
      $S \leftarrow p \cup \{z\}$
      if $S \notin B$ **then**
        $B \leftarrow B \cup \{S\}$
        $v(S) \leftarrow \Phi(p)$
      else
        $v(S) \leftarrow \max(v(S), \Phi(p))$
      end if
    end for
  end for
  $\hat{B} \leftarrow \{S \mid S \in B, \Phi(S) > v(S)\}$ ▷ Filter
  return $\hat{B}$
end procedure

Reducción simplificada

Algorithm 3 Simple reduction

```
procedure REDUCTION(B, PZ | Φ, d)
    L ← list out of B sorted by decreasing value on Φ
    Q ← priority queue of pairs sorted by diss, empty.
    for i = 0 to L do
        for j = i+1 to |L| do
            u ← new pair                          ▷ A pair has fst, snd and diss.
            fst(u) ← L[i]
            snd(u) ← L[j]
            diss(u) ←DISSIMILITUDE(fst(u),snd(u) | d)
            add u to Q
        end for
    end for
    while |L| > PZ do
        u ← extract pair of smaller diss of Q
        if (fst(u) ∈ L) ∧ (snd(u) ∈ L) then
            remove snd(u) from L
        end if
    end while
    return L as a set
end procedure
```

Reducción con heurística-VR

Algorithm 4 Reduction with VR-heuristic

```
procedure REDUCTION(B, PZ, VR | Φ, d)
  L ← list out of B sorted by decreasing value on Φ
  Q ← priority queue of pairs sorted by diss
  for i = 0 to L do
    for j = i+1 to min(i+VR, |L|) do
      u ← new pair
      fst(u) ← L[i]
      snd(u) ← L[j]
      diss(u) ←DISSIMILITUDE(fst(u),snd(u) | d)
      add u to Q
    end for
  end for
  while |L| > PZ do
    u ← extract pair of smaller diss of Q
    if (fst(u) ∈ L) ∧ (snd(u) ∈ L) then
      k ← current index of snd(u) on L
      remove snd(u) from L
      for r = 0 to (VR−1) do
        if (0 ≤ k−VR+r) ∧ (k+r < |L|) then
          f ← new pair
          fst(f) ← L[k−VR+r]
          snd(f) ← L[k+r]
          diss(f) ←DISSIMILITUDE(fst(f),snd(f) | d)
          add f to Q
        end if
      end for
    end if
  end while
  return L as a set
end procedure
```

Descripción del algorimo

Parámetros significativos:

- n: Número de instalaciones.
- m: Número de clientes.
- P: Tamaño de *pool*.
- V: Rango de visión del proceso de reducción.
- p: Tamaño de soluciones buscado.

Se probarán con dos instancias de SPLP de UrlLib: `Euclid` y `GapA`.

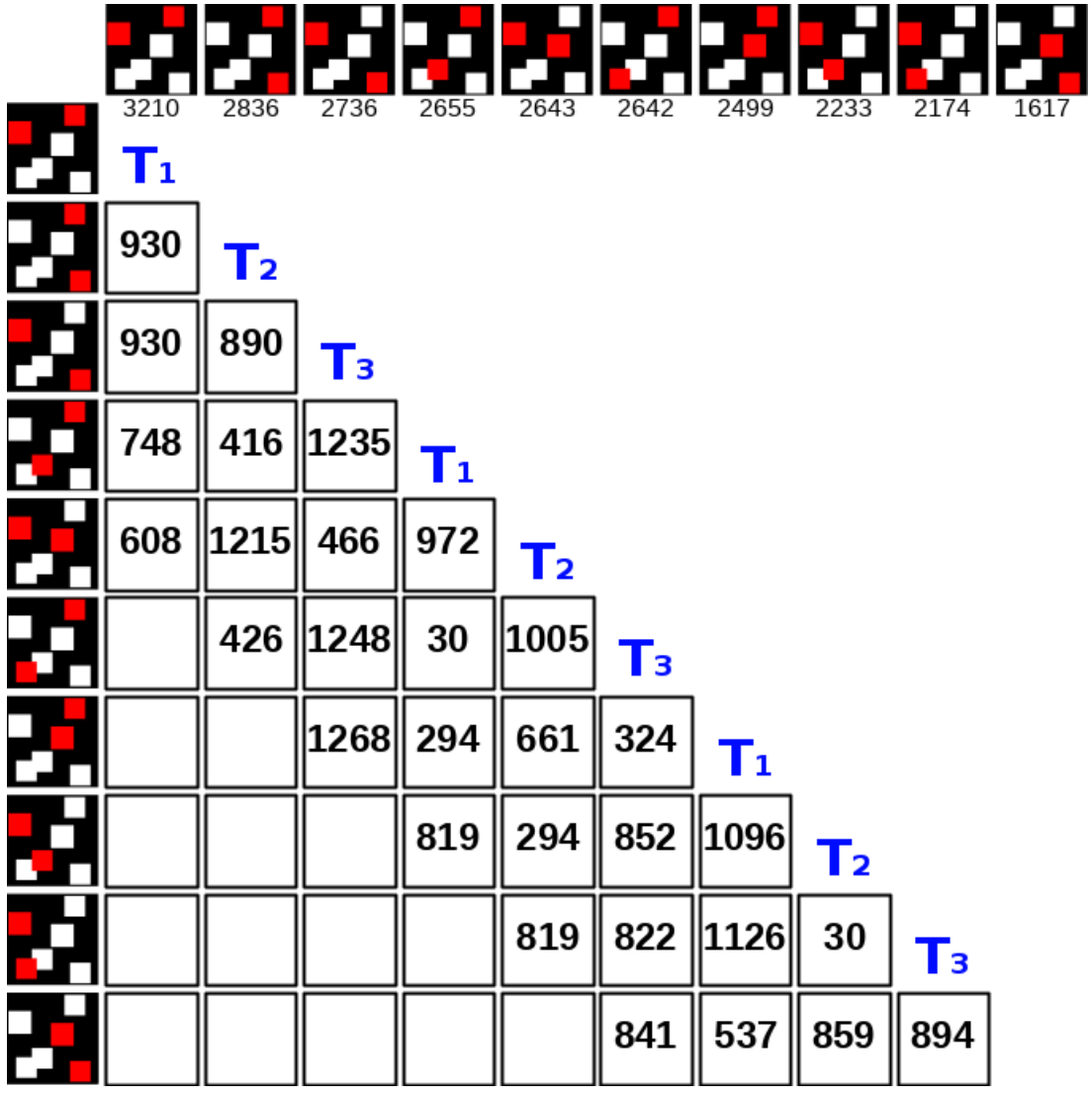
Profiling (problema Euclid)

Profiling (problema GapA)

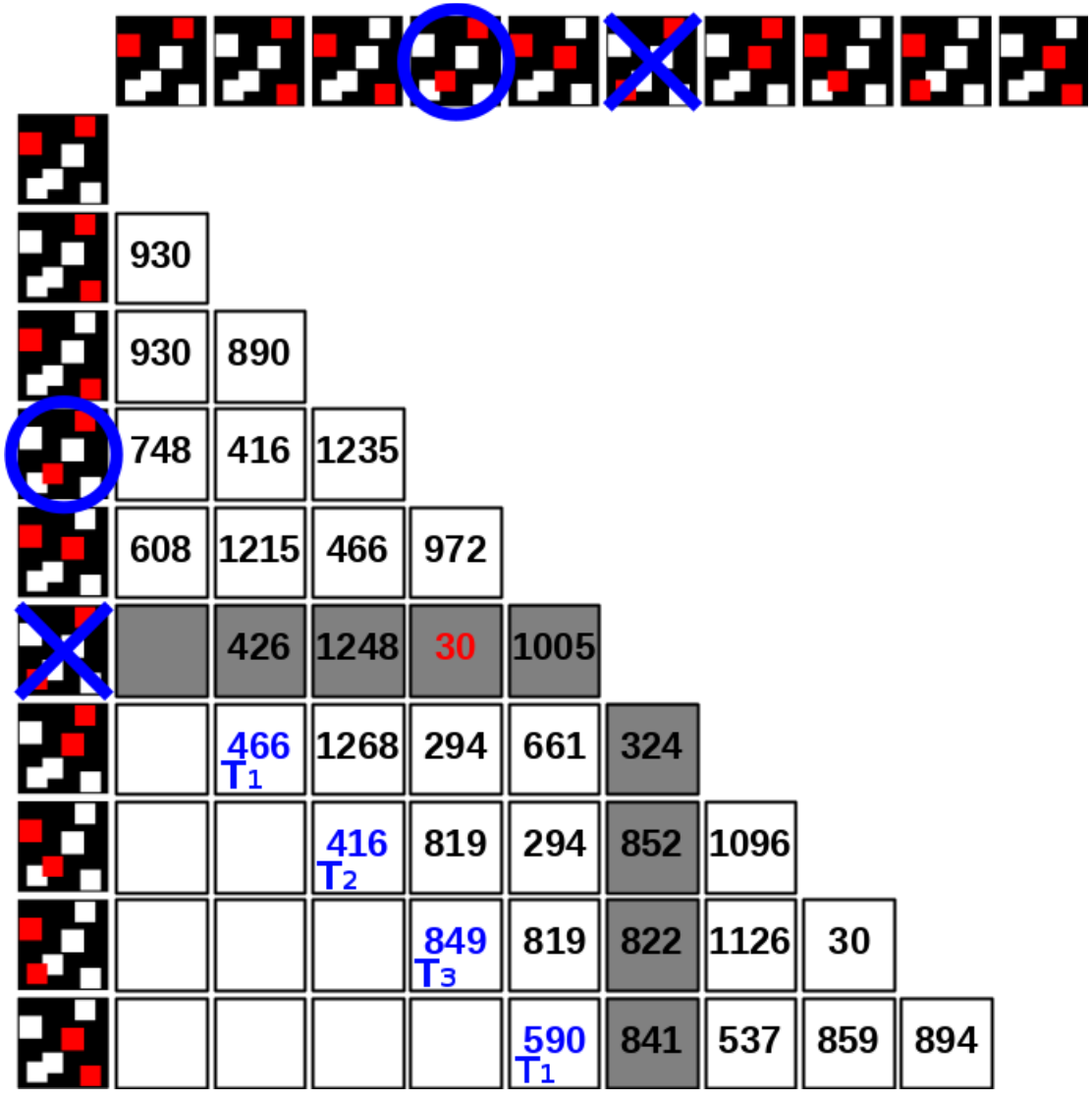
Paralelización de la Reducción

- Según `perf` el principal cuello de botella del algoritmo es el cálculo de la **disimimilitud** entre soluciones.
- Cada una tiene complejidad $O(p^2)$ y en cada proceso de reducción deben realizarse hasta $O(V \cdot P \cdot n)$.
- Adicionalmente cada disimilitud calculada debe ser insertada en un *heap*.

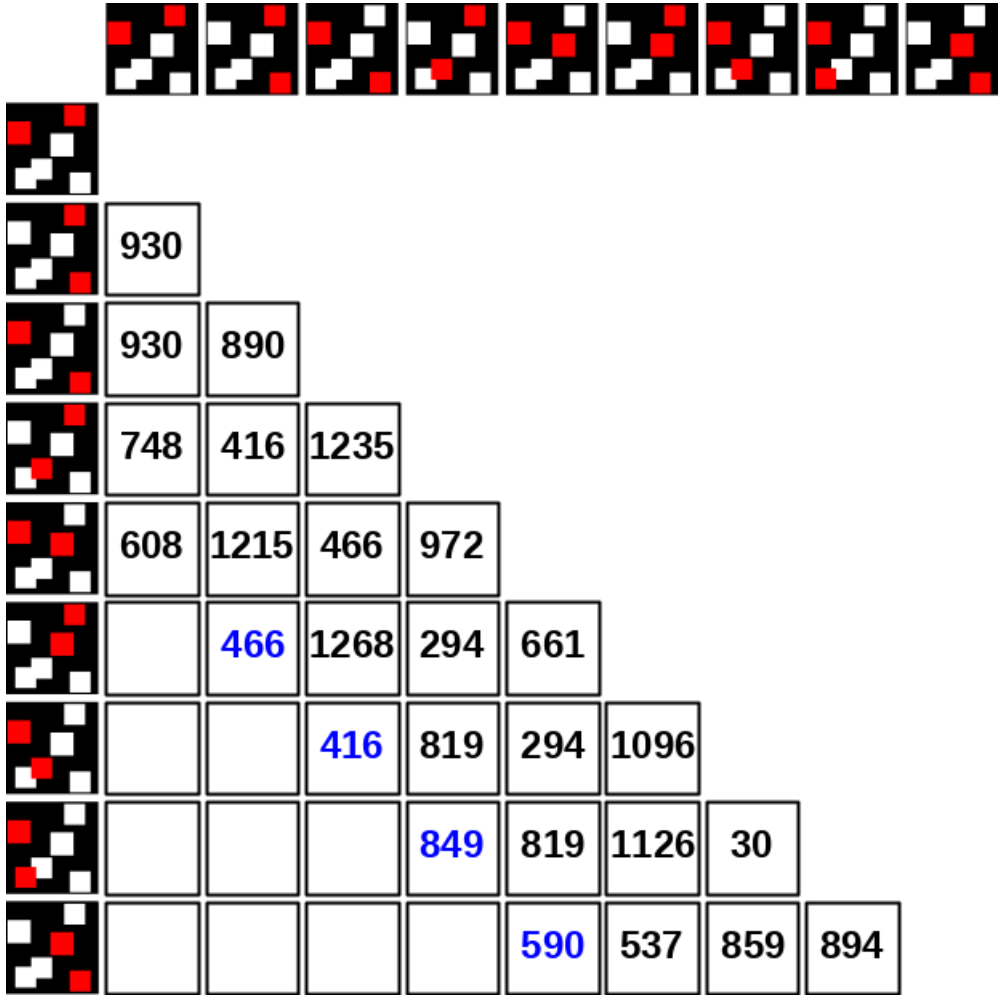
Reducción


3210
2836
2736
2655
2643
2642
2499
2233
2174
1617
T1
930
T2
930
890
T3
748
416
1235
T1
608
1215
466
972
T2
426
1248
30
1005
T3
1268
294
661
324
T1
819
294
852
1096
T2
819
822
1126
30
T3
841
537
859
894

Reducción


930
930 890
748 416 1235
608 1215 466 972
426 1248 30 1005
466 T1 1268 294 661 324
416 T2 819 294 852 1096
849 T3 819 822 1126 30
590 T1 841 537 859 894

Reducción


930
930 890
748 416 1235
608 1215 466 972
466 1268 294 661
416 819 294 1096
849 819 1126 30
590 537 859 894

Paralelización de la Reducción

- Se utilizó un mutex para acceder al *heap*.
- Cada thread computa una ráfaga de V disimilitudes y luego las agrega al heap.
- En las reposiciones se deben agregar V disimilitudes, cada thread agrega V/n, usando `trylock`, si no puede obtener el lock, las almacena en un buffer.
- De esta manera siempre está ocupado el *heap*.

Paralelización de la Reducción

```
void *reduce_thread_execution(void *arg){
  // ...
  for(int i=args->thread_id;i<args->n_sols;i+=THREADS){
    for(int j=1;j<=args->vision_range;j++){
      if(i+j>=args->n_sols) break;
      // -> compute dissimilitude between i and i+j
    }
    pthread_mutex_lock(args->heap_mutex);
    // -> save dissimilitudes in heap
    pthread_mutex_unlock(args->heap_mutex);
    // -> delete dissimilitudes here
  }
  free(pairs);
```

```
while(1){
  sem_post(args->complete_sem);
  // ---@> Main thread works here, read terminated.
  sem_wait(args->thread_sem);
  if(terminated) break;
  // Create new pairs
  for(int i=args->thread_id;i<args->vision_range;
      i+=THREADS){
    // -> compute dissim between prev[vr-1-i] and next[i]
    // -> dissim to buffer
    if(pthread_mutex_trylock(args->heap_mutex)==0){
      // -> add dissims in buffer
      pthread_mutex_unlock(args->heap_mutex);
    }
  }
  // if buffer not empty, wait for mutex
  if(pair_buffer_len>0){
    pthread_mutex_lock(args->heap_mutex);
    // -> add dissims in buffer
    pthread_mutex_unlock(args->heap_mutex);
    pair_buffer_len = 0;
  }
  return NULL;
}
```

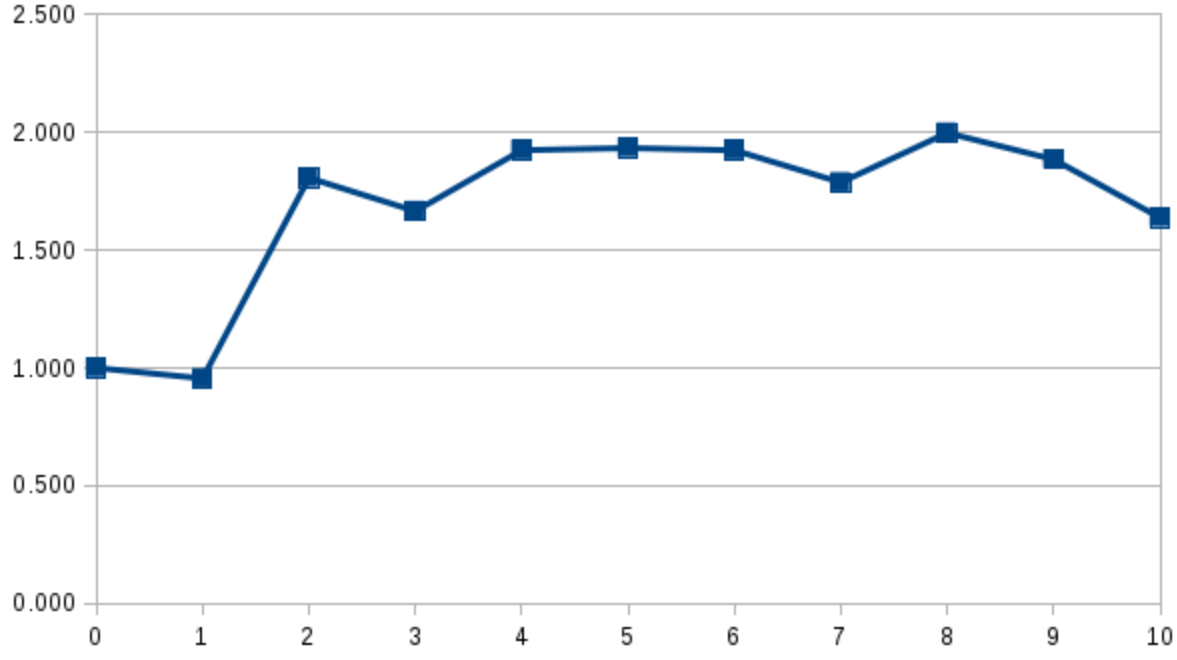
Paralelización de Expansión y Búsqueda Local

- Adicionalmente se paralelizaron estas dos componentes.
- Puesto que se pudo hacer sin necesidad de más concurrencia que el **join** de los threads.

Speedup Euclid

Euclid	CPU[s]	Elapsed[s]	Speedup
0	34.541	34.548	1.000
1	36.101	36.155	0.956
2	36.751	19.121	1.807
3	53.521	20.743	1.666
4	56.110	17.956	1.924
5	51.748	17.870	1.933
6	56.745	17.958	1.924
7	54.394	19.329	1.787
8	55.072	17.296	1.997
9	58.175	18.329	1.885
10	59.084	21.092	1.638

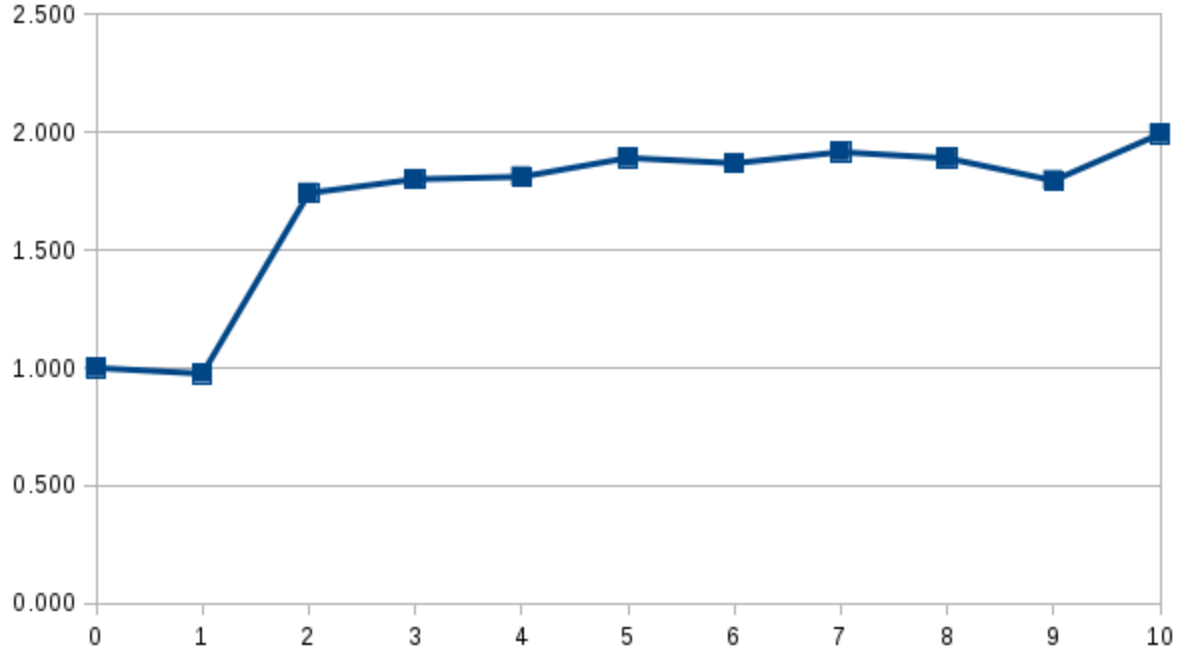
Speedup Euclid



Speedup GapA

Gapa	CPU[s]	Elapsed[s]	Speedup
0	114.678	114.696	1.000
1	116.238	117.559	0.976
2	125.567	65.826	1.742
3	160.371	63.688	1.801
4	186.529	63.314	1.812
5	184.575	60.639	1.891
6	192.403	61.366	1.869
7	194.218	59.861	1.916
8	196.082	60.681	1.890
9	196.667	63.885	1.795
10	195.231	57.557	1.993

Speedup GapA



Conclusiones

- Se puede ver un *speedup* cercano a 2.0 cuando el óptimo es 4.0, lo que puede deberse a las tareas que realiza el *thread* principal.
- El *speed-up* puede ser **mayor** para problemas en que el tamaño de las soluciones y V son mayores, puesto que el tiempo de calcular aumenta en relación con es mayor que el acceso al *heap* y otras tareas.
- La implementación con `trylock` permitió pasar de *speedup* ≈ 1.6 a ≈ 2.0.