TELECOM SudParis

INSTITUT POLYTECHNIQUE DE PARIS

Approximate Pattern Matching

MARIE Nordine & SAVES Marion

Institut Mines-Télécom

Hardware

Hardware

Root :

CPU :

Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz

40 coeurs

GPU :

Quadro P5000 16Go

Other :

CPU :

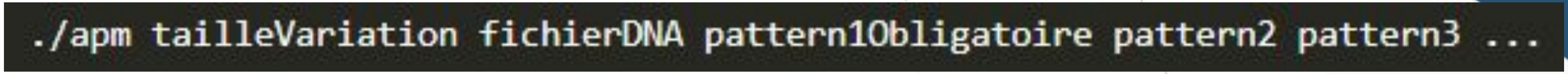
Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz

Structure du code

Structure du code

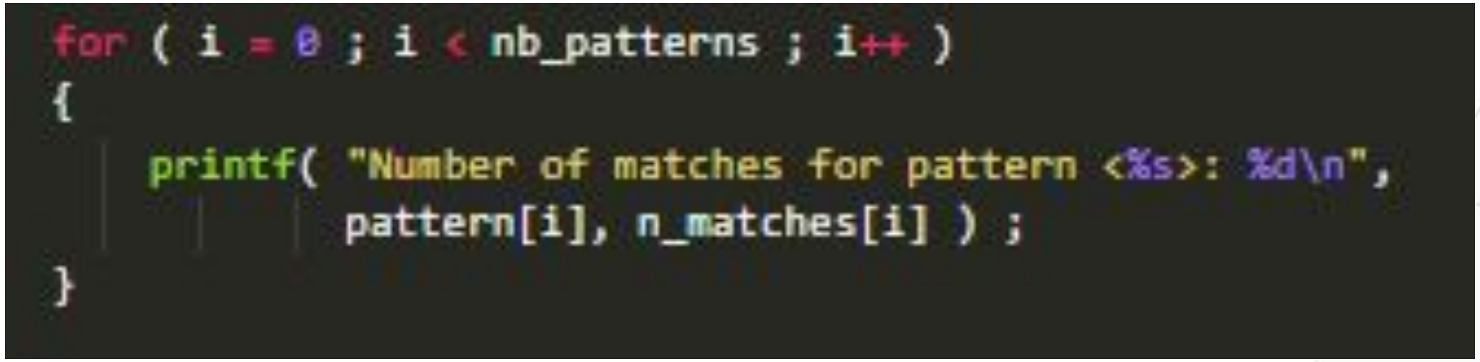
Entrée - Sortie

En entrée :



```
./apm tailleVariation fichierDNA pattern1Obligatoire pattern2 pattern3 ...
```

En sortie :



```
for ( i = 0 ; i < nb_patterns ; i++ )
{
    printf( "Number of matches for pattern <%s>: %d\n",
            pattern[i], n_matches[i] ) ;
}
```

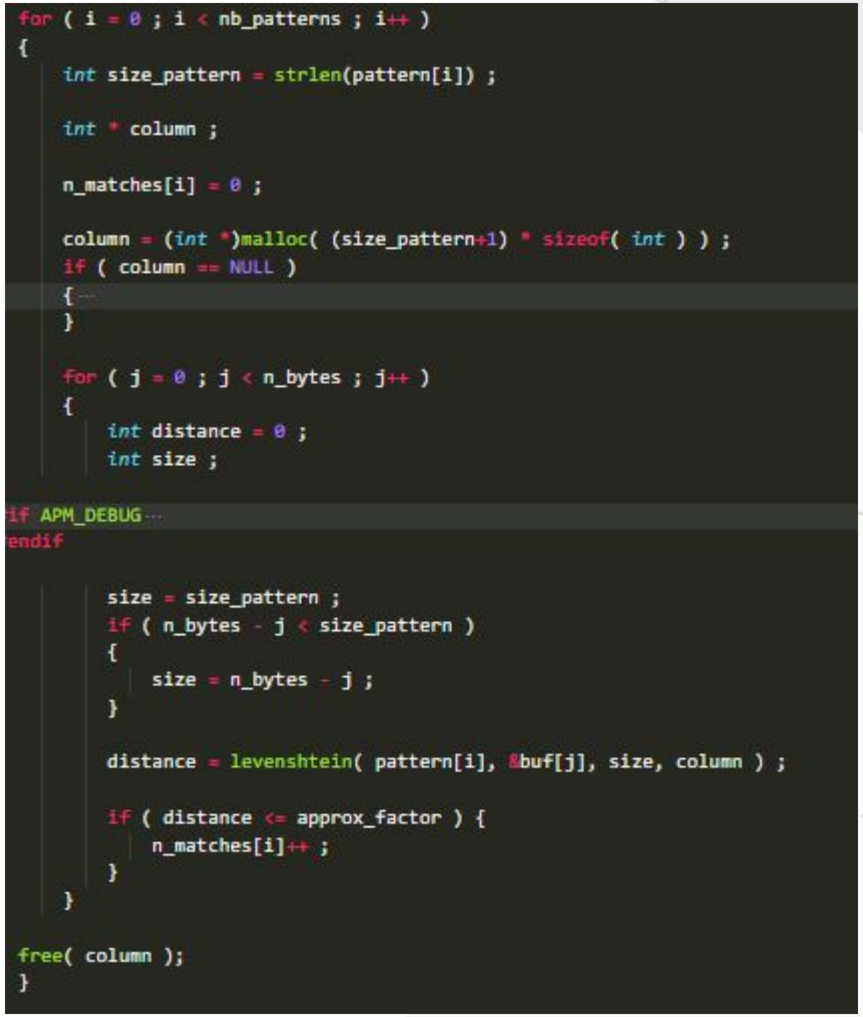
Structure du code

Main

1 - Boucle pour chaque pattern

2 - Boucle pour chaque lettre de l'ADN

3- Appel levenshtein



```
for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
    if ( column == NULL )
    {···
    }

    for ( j = 0 ; j < n_bytes ; j++ )
    {
        int distance = 0 ;
        int size ;

if APM_DEBUG···
endif

        size = size_pattern ;
        if ( n_bytes - j < size_pattern )
        {
            size = n_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            n_matches[i]++ ;
        }
    }

free( column );
}
```

Structure du code

Levenshtein

Pattern :
AG

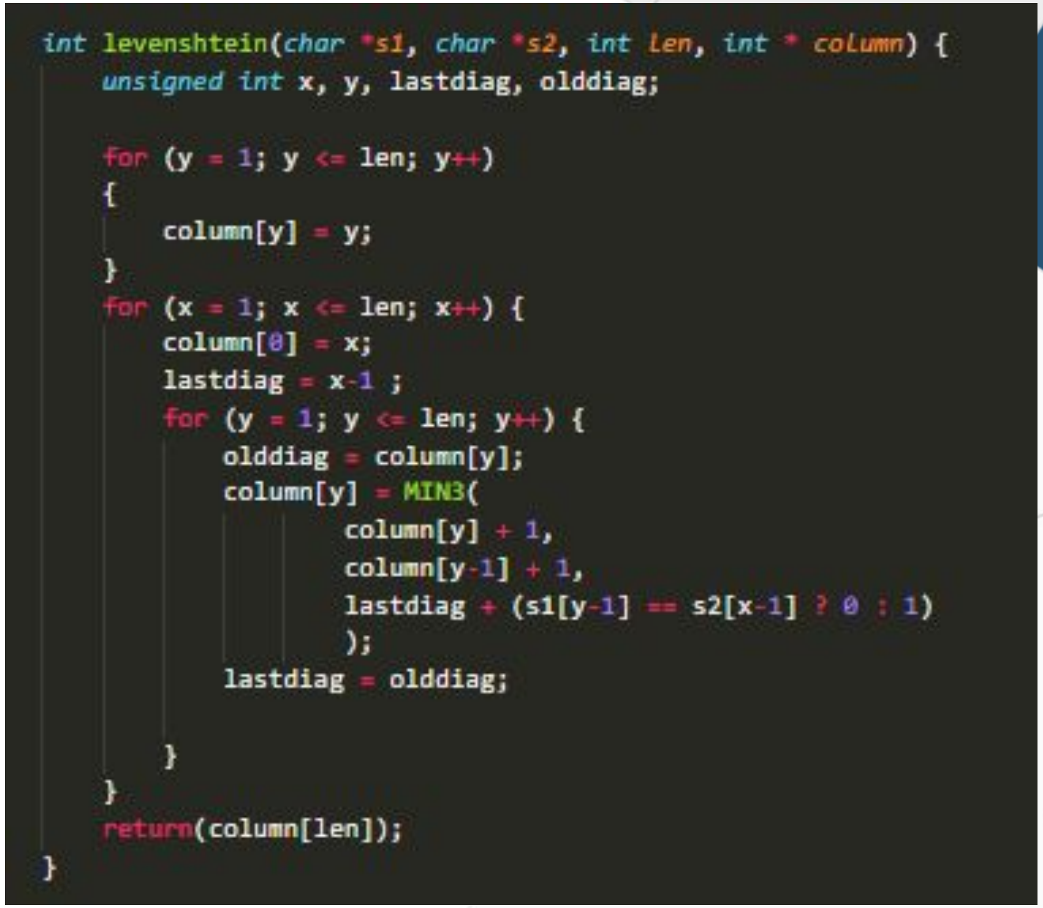
ADN:
ACGT

Levenshtein :
ACGT -> AG et AC distance 1

A**C**GT -> AG et CG distance 1
Ç
AC**G**T -> AG et GT distance 2

ACG**T**

Si tolérence 0 -> 0 fois
Si tolérence 1 -> 2 fois
Si tolérence =>2 -> 3 fois



```c
int levenshtein(char *s1, char *s2, int len, int * column) {
    unsigned int x, y, lastdiag, olddiag;

    for (y = 1; y <= len; y++)
    {
        column[y] = y;
    }
    for (x = 1; x <= len; x++) {
        column[0] = x;
        lastdiag = x-1 ;
        for (y = 1; y <= len; y++) {
            olddiag = column[y];
            column[y] = MIN3(
                    column[y] + 1,
                    column[y-1] + 1,
                    lastdiag + (s1[y-1] == s2[x-1] ? 0 : 1)
                    );
            lastdiag = olddiag;

        }
    }
    return(column[len]);
}
```

TELECOM SudParis

INSTITUT POLYTECHNIQUE DE PARIS

Conditions de test

Institut Mines-Télécom

Conditions de test

DataBase :
chr1_KI270763v1_alt.fa

Distance :
5

Small Pattern :
32 caractères

Medium Pattern:
96 caractères (3 fois small)

Large Pattern :
224 caractères (7 fois small)

Average:
Moyenne pondérée
= (small + 3medium + 7large)/11

TELECOM SudParis

INSTITUT POLYTECHNIQUE DE PARIS

OpenMP

Institut Mines-Télécom

OpenMP

Idées non abouties

Idée 1 :
Chaque thread va s’occuper d’un pattern (stratégie dynamique)

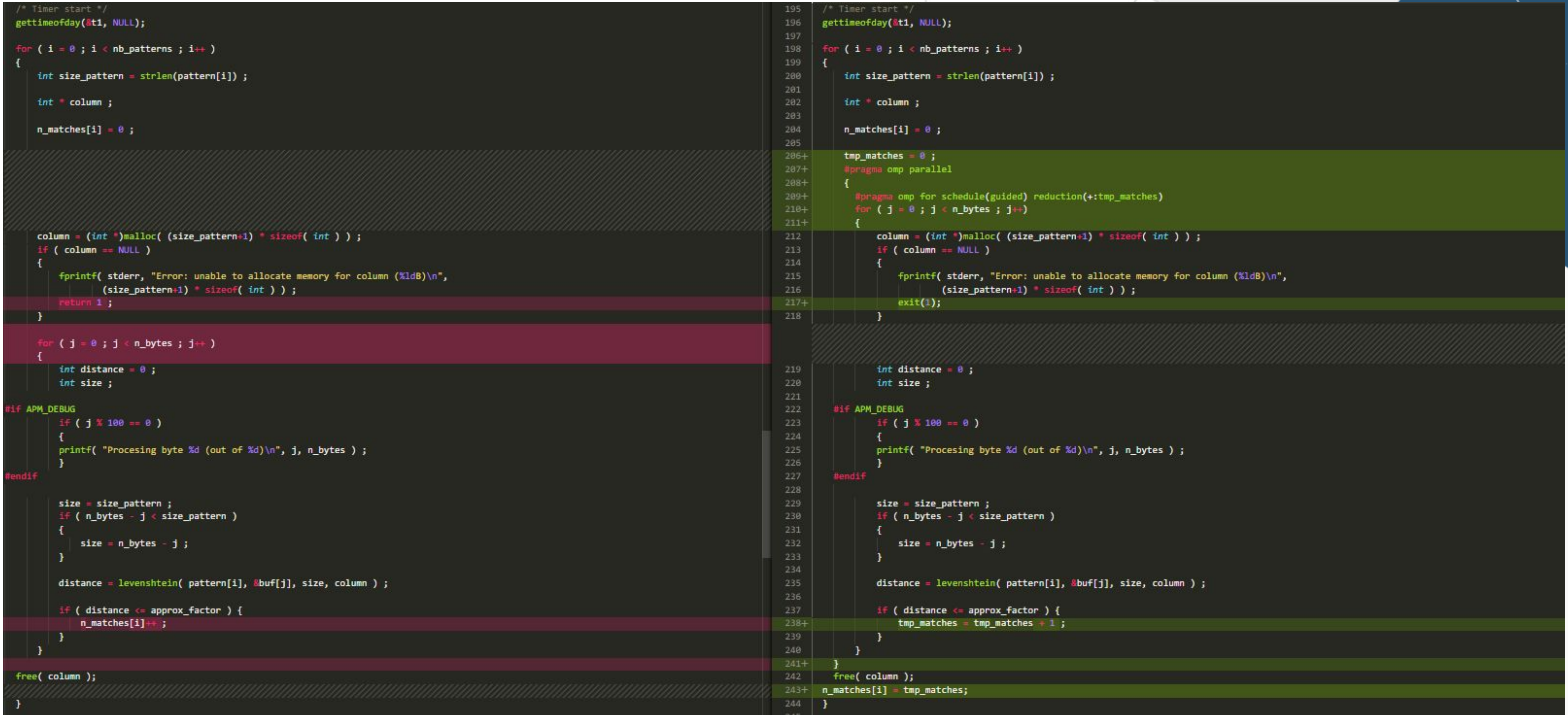
Problème : Déséquilibre se crée

Idée 2:
Chaque thread va s’occuper d’un élément de l’ADN

Problème : Résultat faux, peu d’amélioration → Problème de partage de mémoire

OpenMP

Idée retenues

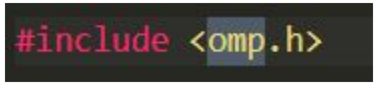


```c
/* Timer start */
gettimeofday(&t1, NULL);

for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
    if ( column == NULL )
    {
        fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
                (size_pattern+1) * sizeof( int ) ) ;
        return 1 ;
    }

    for ( j = 0 ; j < n_bytes ; j++ )
    {
        int distance = 0 ;
        int size ;

#if APM_DEBUG
        if ( j % 100 == 0 )
        {
        printf( "Procesing byte %d (out of %d)\n", j, n_bytes ) ;
        }
#endif

        size = size_pattern ;
        if ( n_bytes - j < size_pattern )
        {
            size = n_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            n_matches[i]++ ;
        }
    }

free( column );

}
```

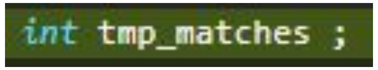
```c
/* Timer start */
gettimeofday(&t1, NULL);

for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    tmp_matches = 0 ;
    #pragma omp parallel
    {
      #pragma omp for schedule(guided) reduction(+:tmp_matches)
      for ( j = 0 ; j < n_bytes ; j++)
      {
        column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
        if ( column == NULL )
        {
            fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
                    (size_pattern+1) * sizeof( int ) ) ;
            exit(1);
        }

        int distance = 0 ;
        int size ;

  #if APM_DEBUG
        if ( j % 100 == 0 )
        {
        printf( "Procesing byte %d (out of %d)\n", j, n_bytes ) ;
        }
  #endif

        size = size_pattern ;
        if ( n_bytes - j < size_pattern )
        {
            size = n_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            tmp_matches = tmp_matches + 1 ;
        }
      }
  }
  free( column );
n_matches[i] = tmp_matches;
}
```

OpenMP

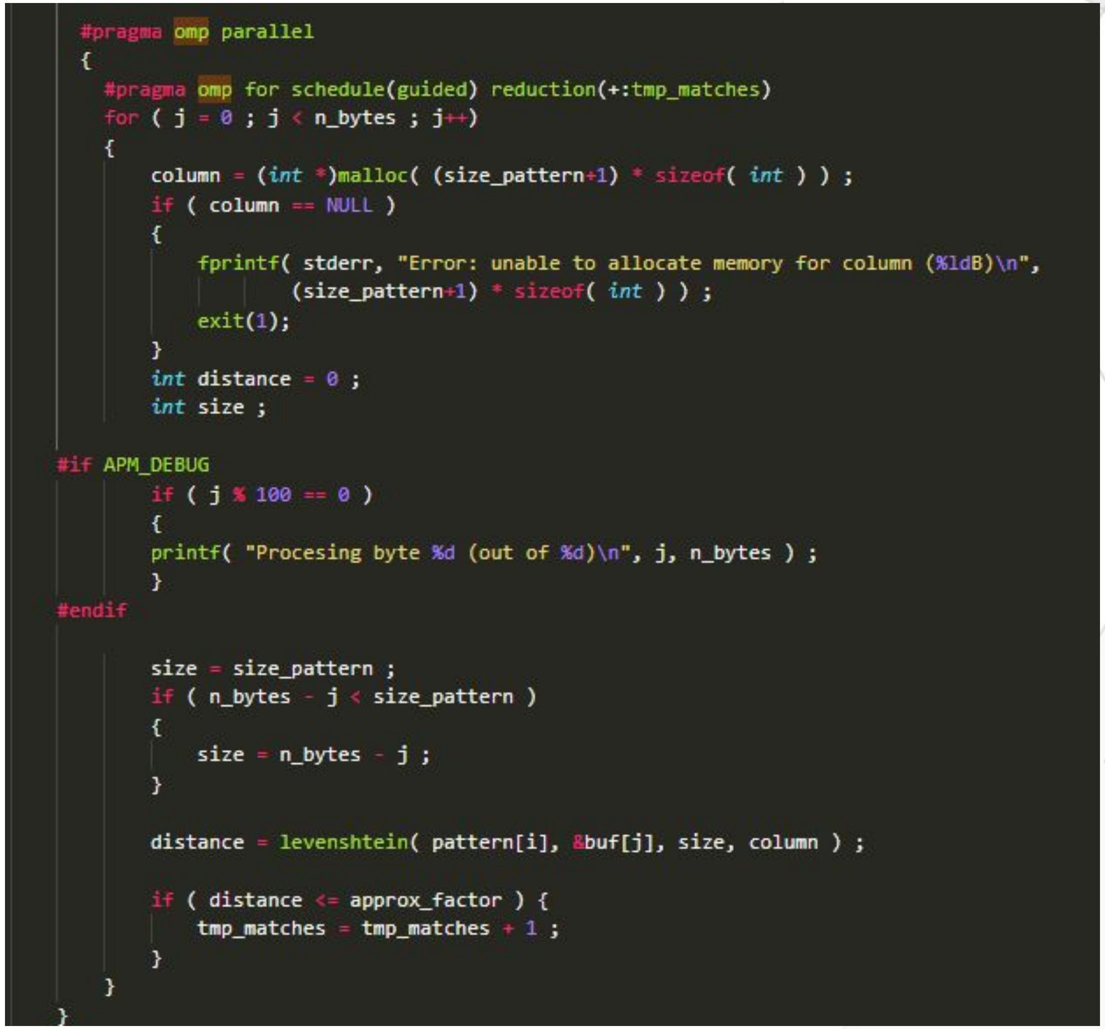
Idée retenues



```
#include <omp.h>
```



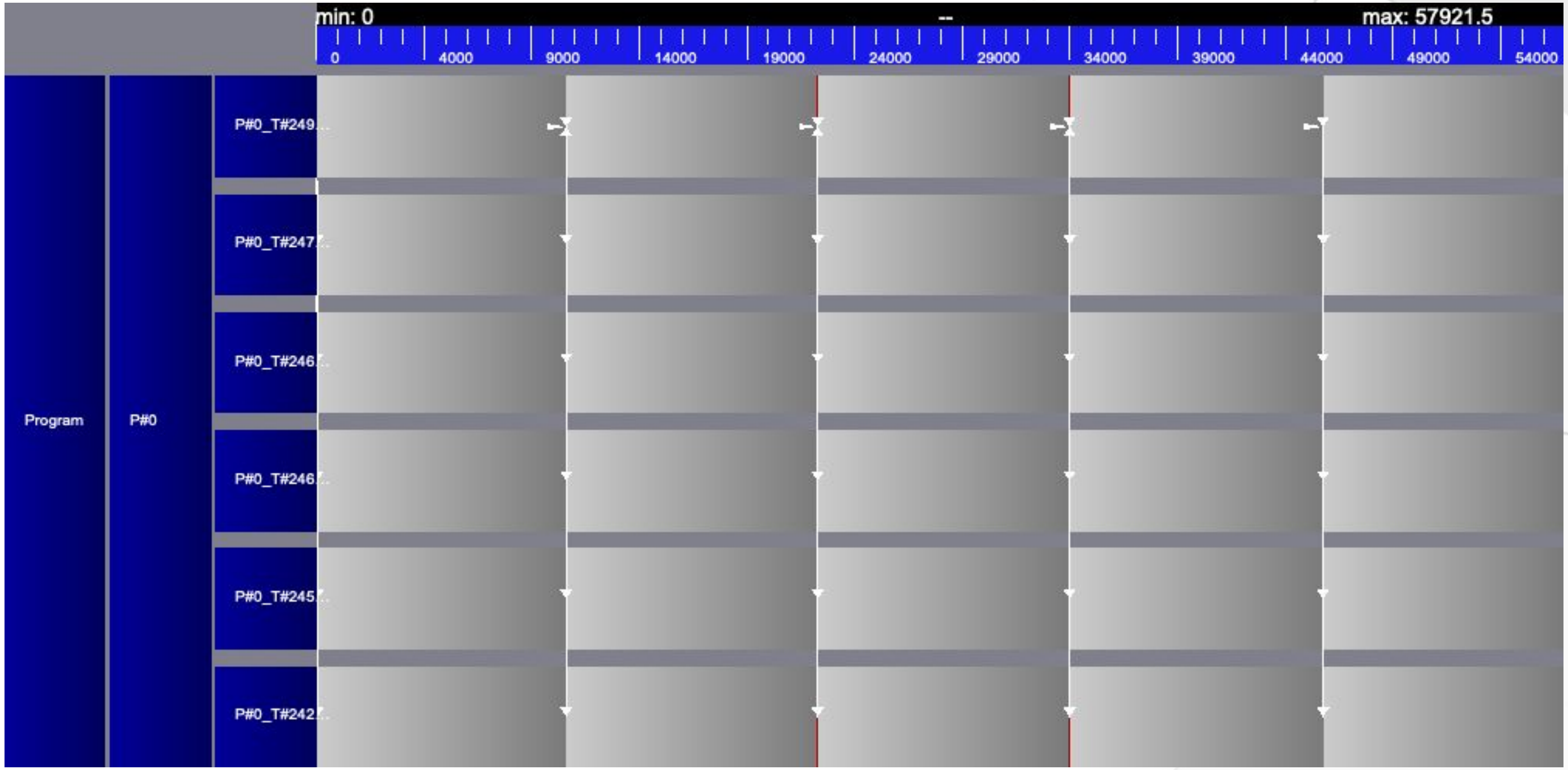
```
int tmp_matches ;
```



```
 #pragma omp parallel
 {
  #pragma omp for schedule(guided) reduction(+:tmp_matches)
  for ( j = 0 ; j < n_bytes ; j++)
  {
    column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
    if ( column == NULL )
    {
      fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
          (size_pattern+1) * sizeof( int ) ) ;
      exit(1);
    }
    int distance = 0 ;
    int size ;

#if APM_DEBUG
    if ( j % 100 == 0 )
    {
    printf( "Procesing byte %d (out of %d)\n", j, n_bytes ) ;
    }
#endif

    size = size_pattern ;
    if ( n_bytes - j < size_pattern )
    {
      size = n_bytes - j ;
    }

    distance = levenshtein( pattern[i], &buf[j], size, column ) ;

    if ( distance <= approx_factor ) {
      tmp_matches = tmp_matches + 1 ;
    }
  }
}
```

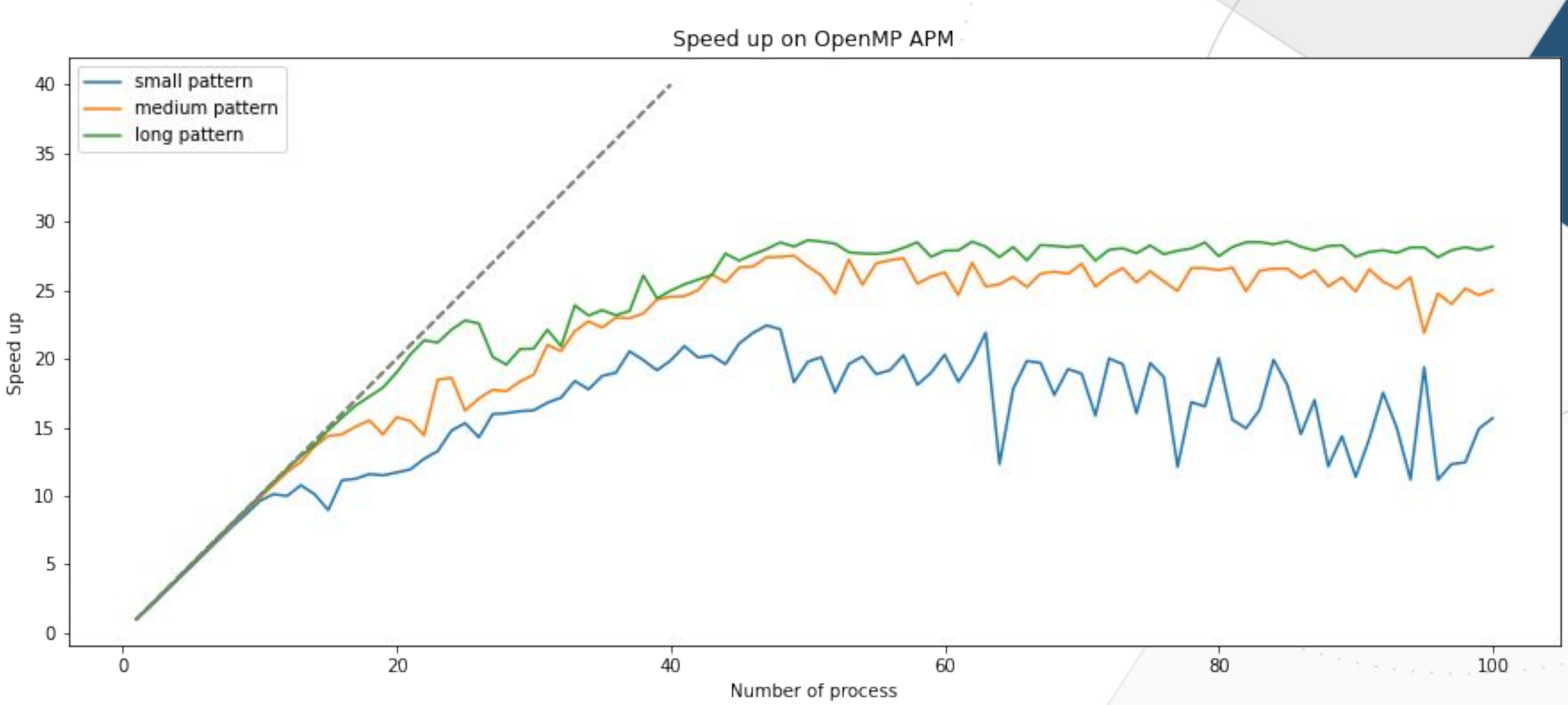
OpenMP

Trace



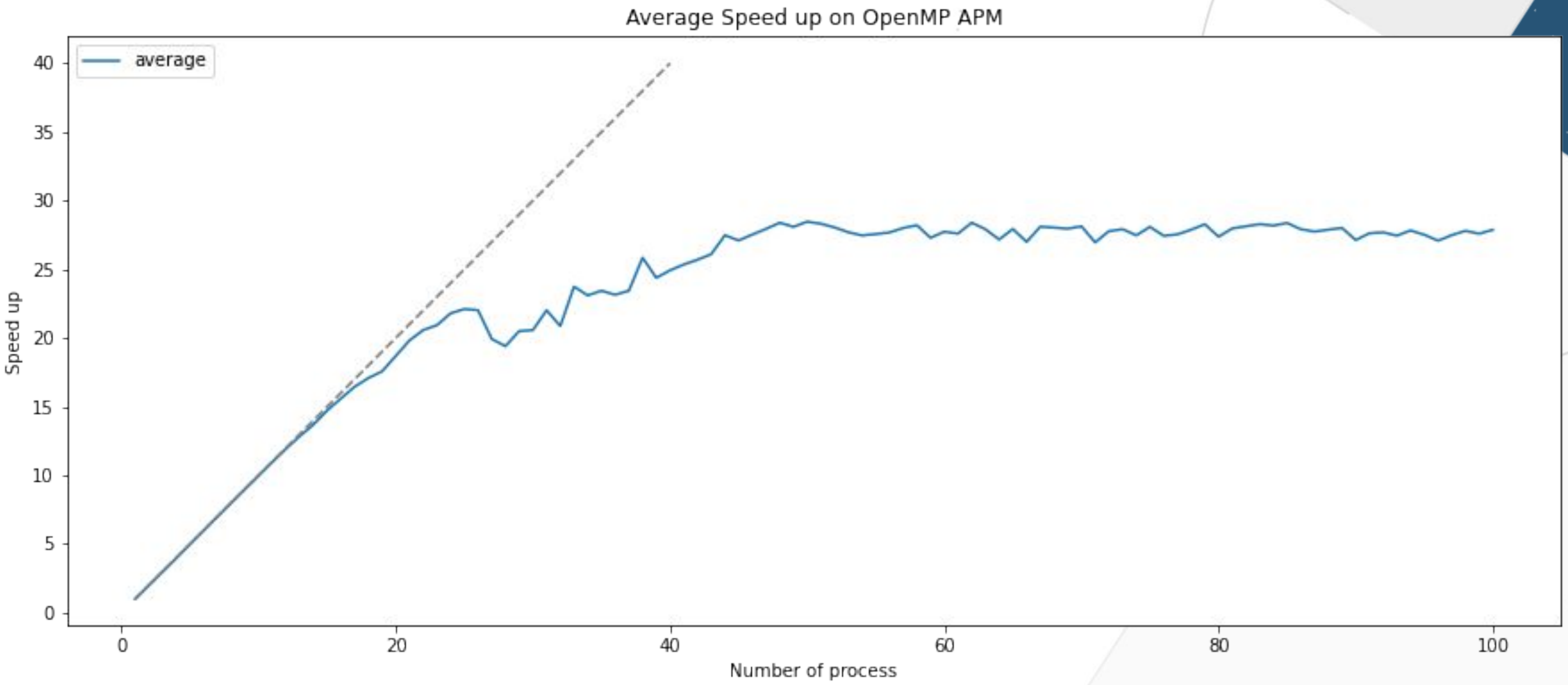
OpenMP

SpeedUp



OpenMP

SpeedUp

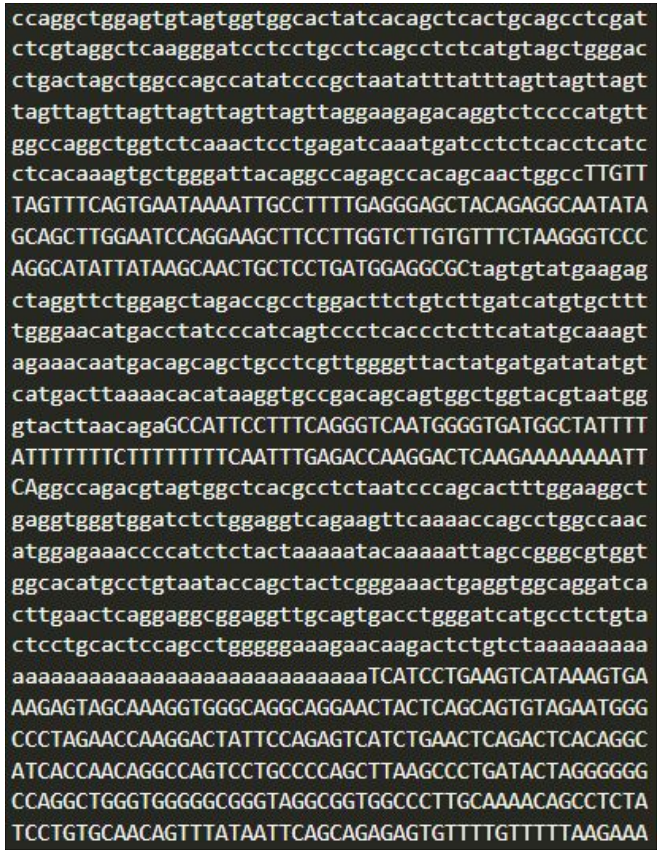


MPI

MPI

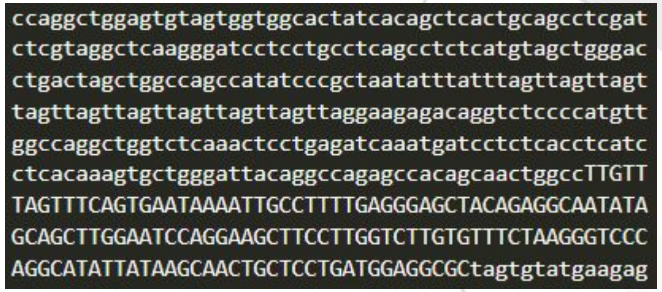
Idée générale

Fichier d’ADN sur une machine

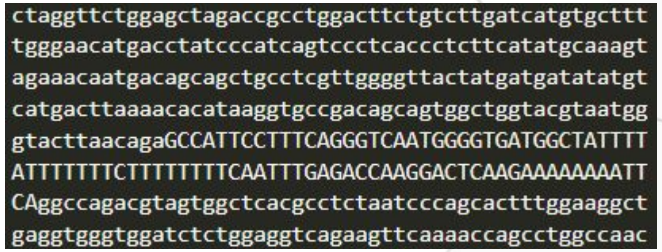


```
ccaggctggagtgtagtggtggcactatcacagctcactgcagcctcgat
ctcgtaggctcaagggatcctcctgcctcagcctctcatgtagctgggac
ctgactagctggccagccatatcccgctaatatttatttagttagttagt
tagttagttagttagttagttaggaagagacaggtctccccatgtt
ggccaggctggtctcaaactcctgagatcaaatgatcctctcacctcatc
ctcacaaagtgctgggattacaggccagagccacagcaactggccTTGTT
TAGTTTCAGTGAATAAAATTGCCTTTTGAGGGAGCTACAGAGGCAATATA
GCAGCTTGGAATCCAGGAAGCTTCCTTGGTCTTGTGTTTCTAAGGGTCCC
AGGCATATTATAAGCAACTGCTCCTGATGGAGGCGCtagtgtatgaagag
ctaggttctggagctagaccgcctggacttctgtcttgatcatgtgcttt
tgggaacatgacctatcccatcagtccctcaccctcttcatatgcaaagt
agaaacaatgacagcagctgcctcgttggggttactatgatgatatatgt
catgacttaaaacacataaggtgccgacagcagtggctggtacgtaatgg
gtacttaacagaGCCATTCCTTTCAGGGTCAATGGGGTGATGGCTATTTT
ATTTTTTTCTTTTTTTTTCAATTTGAGACCAAGGACTCAAGAAAAAAATT
CAggccagacgtagtggctcacgcctctaatcccagcactttggaaggct
gaggtgggtggatctctggaggtcagaagttcaaaaccagcctggccaac
atggagaaaccccatctctactaaaaatacaaaaattagccgggcgtggt
ggcacatgcctgtaataccagctactcgggaaactgaggtggcaggatca
cttgaactcaggaggcggaggttgcagtgacctgggatcatgcctctgta
ctcctgcactccagcctgggggaaagaacaagactctgtctaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaTCATCCTGAAGTCATAAAGTGA
AAGAGTAGCAAAGGTGGGCAGGCAGGAACTACTCAGCAGTGTAGAATGGG
CCCTAGAACCAAGGACTATTCCAGAGTCATCTGAACTCAGACTCACAGGC
ATCACCAACAGGCCAGTCCTGCCCCAGCTTAAGCCCTGATACTAGGGGGG
CCAGGCTGGGTGGGGGCGGGTAGGCGGTGGCCCTTGCAAAACAGCCTCTA
TCCTGTGCAACAGTTTATAATTCAGCAGAGAGTGTTTTGTTTTTAAGAAA
```

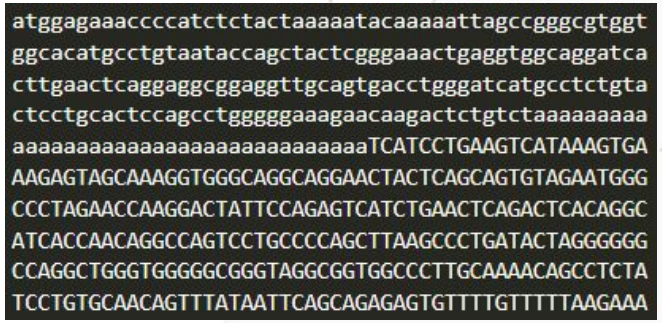
Fichier d’ADN sur trois machines



```
ccaggctggagtgtagtggtggcactatcacagctcactgcagcctcgat
ctcgtaggctcaagggatcctcctgcctcagcctctcatgtagctgggac
ctgactagctggccagccatatcccgctaatatttatttagttagttagt
tagttagttagttagttagttaggaagagacaggtctccccatgtt
ggccaggctggtctcaaactcctgagatcaaatgatcctctcacctcatc
ctcacaaagtgctgggattacaggccagagccacagcaactggccTTGTT
TAGTTTCAGTGAATAAAATTGCCTTTTGAGGGAGCTACAGAGGCAATATA
GCAGCTTGGAATCCAGGAAGCTTCCTTGGTCTTGTGTTTCTAAGGGTCCC
AGGCATATTATAAGCAACTGCTCCTGATGGAGGCGCtagtgtatgaagag
```



```
ctaggttctggagctagaccgcctggacttctgtcttgatcatgtgcttt
tgggaacatgacctatcccatcagtccctcaccctcttcatatgcaaagt
agaaacaatgacagcagctgcctcgttggggttactatgatgatatatgt
catgacttaaaacacataaggtgccgacagcagtggctggtacgtaatgg
gtacttaacagaGCCATTCCTTTCAGGGTCAATGGGGTGATGGCTATTTT
ATTTTTTTCTTTTTTTTTCAATTTGAGACCAAGGACTCAAGAAAAAAATT
CAggccagacgtagtggctcacgcctctaatcccagcactttggaaggct
gaggtgggtggatctctggaggtcagaagttcaaaaccagcctggccaac
```



```
atggagaaaccccatctctactaaaaatacaaaaattagccgggcgtggt
ggcacatgcctgtaataccagctactcgggaaactgaggtggcaggatca
cttgaactcaggaggcggaggttgcagtgacctgggatcatgcctctgta
ctcctgcactccagcctgggggaaagaacaagactctgtctaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaTCATCCTGAAGTCATAAAGTGA
AAGAGTAGCAAAGGTGGGCAGGCAGGAACTACTCAGCAGTGTAGAATGGG
CCCTAGAACCAAGGACTATTCCAGAGTCATCTGAACTCAGACTCACAGGC
ATCACCAACAGGCCAGTCCTGCCCCAGCTTAAGCCCTGATACTAGGGGGG
CCAGGCTGGGTGGGGGCGGGTAGGCGGTGGCCCTTGCAAAACAGCCTCTA
TCCTGTGCAACAGTTTATAATTCAGCAGAGAGTGTTTTGTTTTTAAGAAA
```

MPI

Difficultées : Effets de bord

Pattern :
CTAG

Sur une machine:
AGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT

Sur deux machines
AGCTAGCTAGCTAGCT AGCTAGCTAGCTAGCT

MPI

Difficultées : Effets de bord

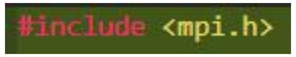
Pattern :
CTAG

Sur une machine:
AGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT

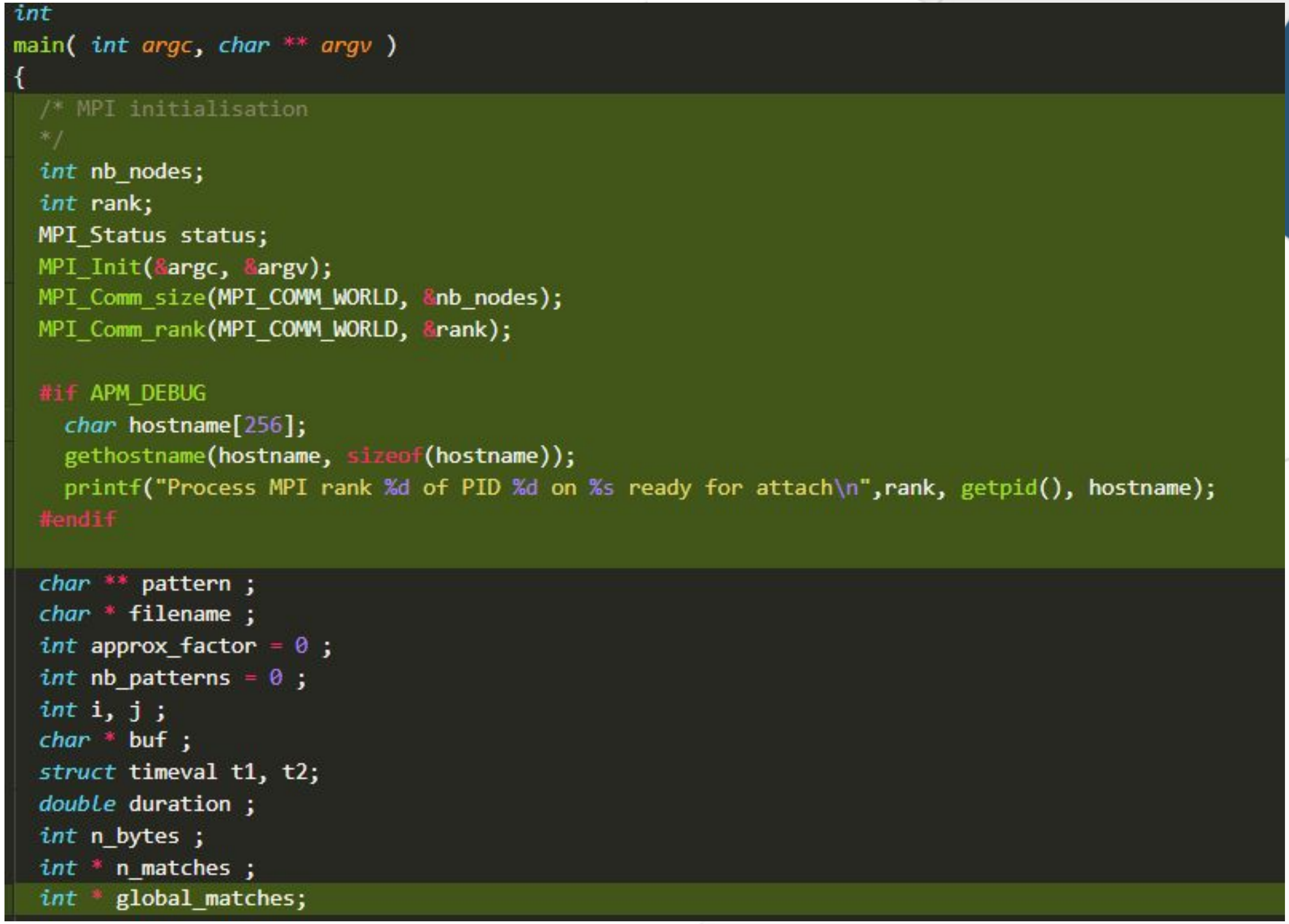
Sur deux machines
AGCTAGCTAGCTAGCT AGCTAGCTAGCTAGCT

MPI

Initialisation



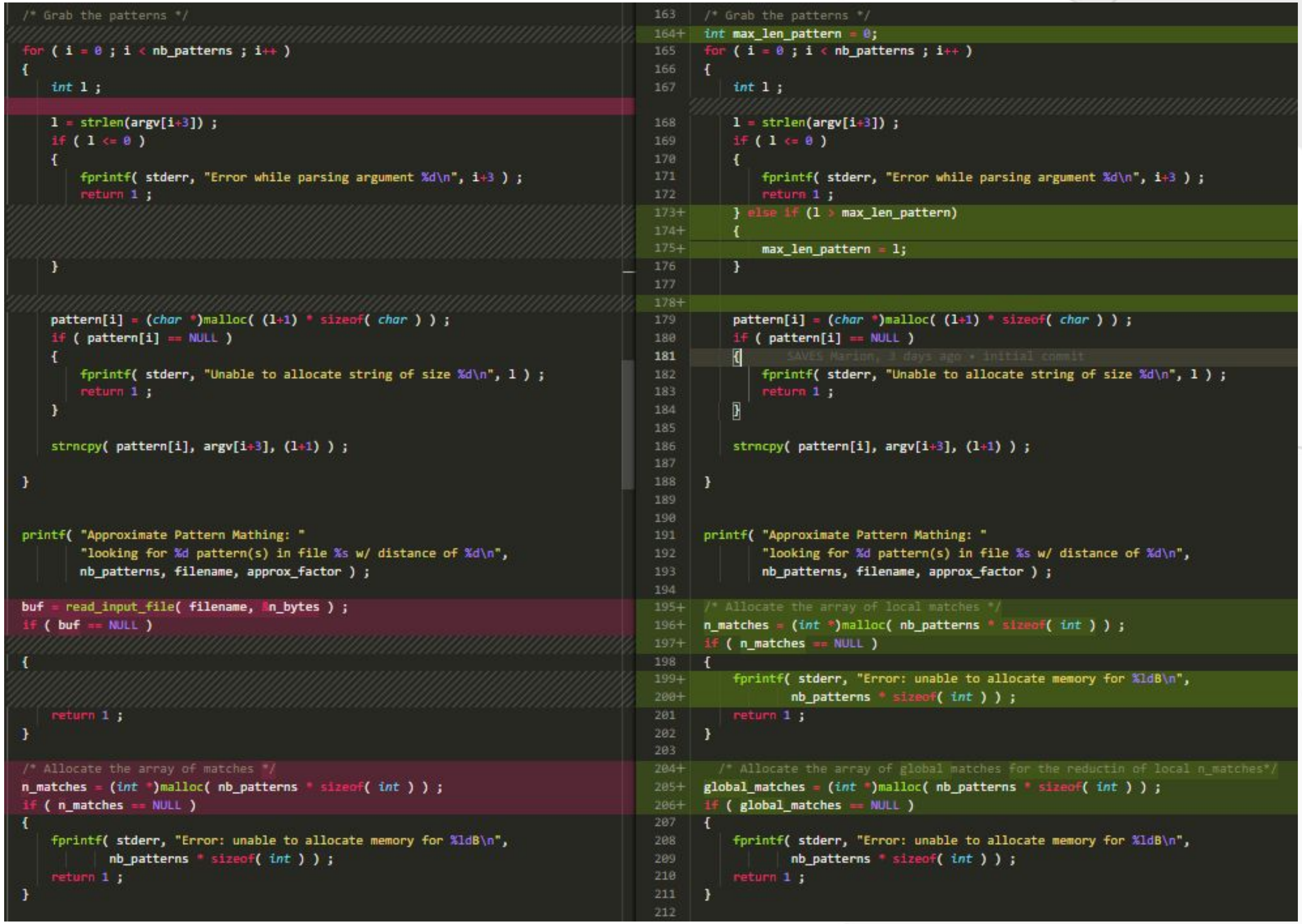
```c
#include <mpi.h>
```



```c
int
main( int argc, char ** argv )
{
  /* MPI initialisation
  */
  int nb_nodes;
  int rank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nb_nodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  #if APM_DEBUG
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    printf("Process MPI rank %d of PID %d on %s ready for attach\n",rank, getpid(), hostname);
  #endif

  char ** pattern ;
  char * filename ;
  int approx_factor = 0 ;
  int nb_patterns = 0 ;
  int i, j ;
  char * buf ;
  struct timeval t1, t2;
  double duration ;
  int n_bytes ;
  int * n_matches ;
  int * global_matches;
```

MPI

Main

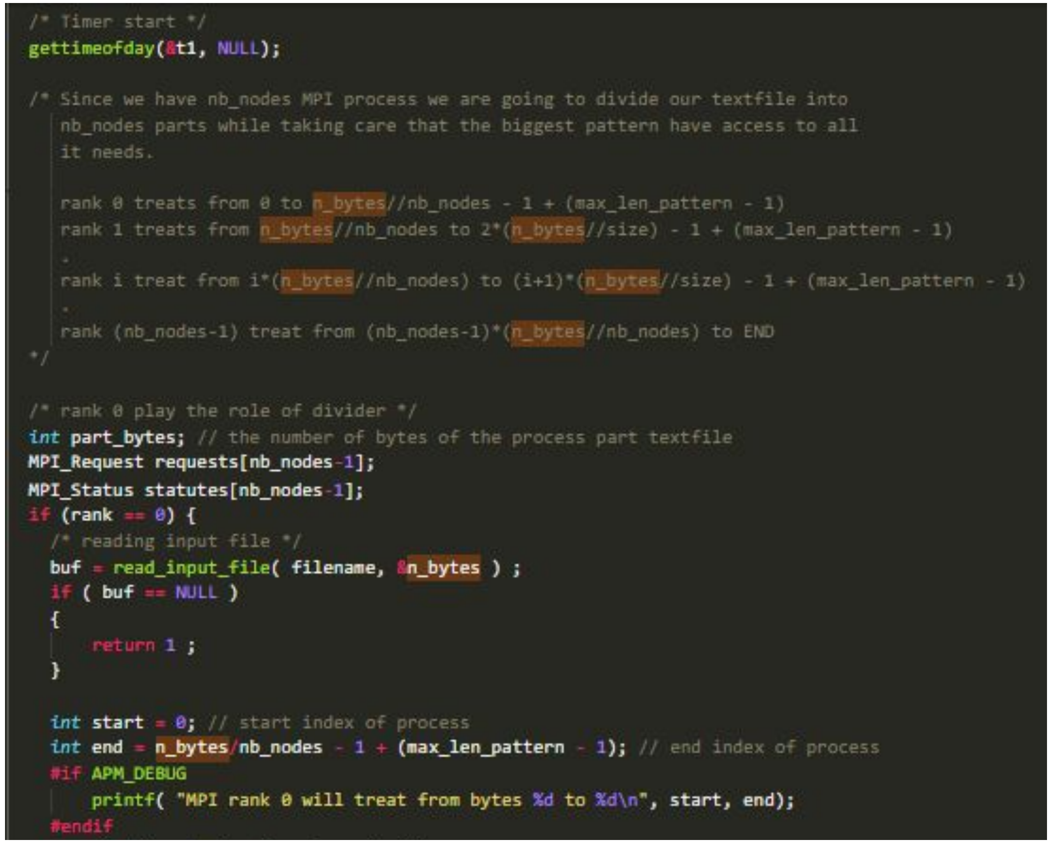


```
/* Grab the patterns */

for ( i = 0 ; i < nb_patterns ; i++ )
{
    int l ;

    l = strlen(argv[i+3]) ;
    if ( l <= 0 )
    {
        fprintf( stderr, "Error while parsing argument %d\n", i+3 ) ;
        return 1 ;



    }


    pattern[i] = (char *)malloc( (l+1) * sizeof( char ) ) ;
    if ( pattern[i] == NULL )
    {
        fprintf( stderr, "Unable to allocate string of size %d\n", l ) ;
        return 1 ;
    }

    strncpy( pattern[i], argv[i+3], (l+1) ) ;

}


printf( "Approximate Pattern Mathing: "
        "looking for %d pattern(s) in file %s w/ distance of %d\n",
        nb_patterns, filename, approx_factor ) ;

buf = read_input_file( filename, &n_bytes ) ;
if ( buf == NULL )

{


    return 1 ;
}

/* Allocate the array of matches */
n_matches = (int *)malloc( nb_patterns * sizeof( int ) ) ;
if ( n_matches == NULL )
{
    fprintf( stderr, "Error: unable to allocate memory for %ldB\n",
            nb_patterns * sizeof( int ) ) ;
    return 1 ;
}
```

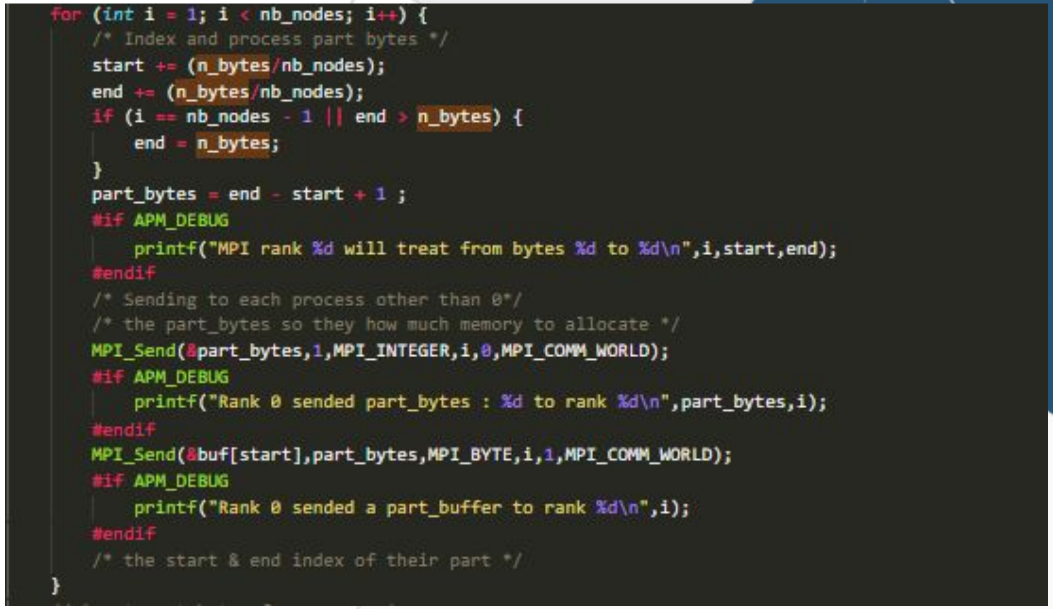
```
/* Grab the patterns */
int max_len_pattern = 0;
for ( i = 0 ; i < nb_patterns ; i++ )
{
    int l ;

    l = strlen(argv[i+3]) ;
    if ( l <= 0 )
    {
        fprintf( stderr, "Error while parsing argument %d\n", i+3 ) ;
        return 1 ;
    } else if (l > max_len_pattern)
    {
        max_len_pattern = l;
    }


    pattern[i] = (char *)malloc( (l+1) * sizeof( char ) ) ;
    if ( pattern[i] == NULL )
    {        SAVES Marion, 3 days ago • initial commit
        fprintf( stderr, "Unable to allocate string of size %d\n", l ) ;
        return 1 ;
    }

    strncpy( pattern[i], argv[i+3], (l+1) ) ;

}


printf( "Approximate Pattern Mathing: "
        "looking for %d pattern(s) in file %s w/ distance of %d\n",
        nb_patterns, filename, approx_factor ) ;

/* Allocate the array of local matches */
n_matches = (int *)malloc( nb_patterns * sizeof( int ) ) ;
if ( n_matches == NULL )
{
    fprintf( stderr, "Error: unable to allocate memory for %ldB\n",
            nb_patterns * sizeof( int ) ) ;
    return 1 ;
}

  /* Allocate the array of global matches for the reductin of local n_matches*/
global_matches = (int *)malloc( nb_patterns * sizeof( int ) ) ;
if ( global_matches == NULL )
{
    fprintf( stderr, "Error: unable to allocate memory for %ldB\n",
            nb_patterns * sizeof( int ) ) ;
    return 1 ;
}

```

MPI

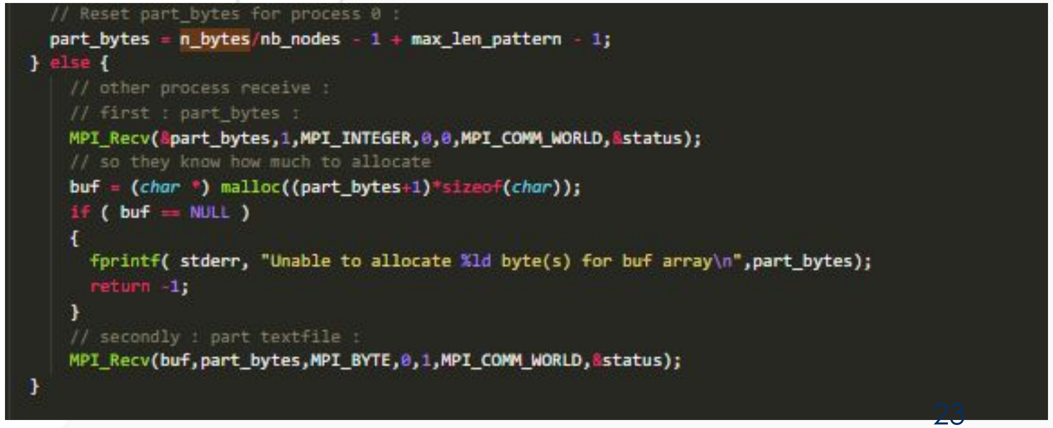
Main



```
/* Timer start */
gettimeofday(&t1, NULL);

/* Since we have nb_nodes MPI process we are going to divide our textfile into
   nb_nodes parts while taking care that the biggest pattern have access to all
   it needs.

   rank 0 treats from 0 to n_bytes//nb_nodes - 1 + (max_len_pattern - 1)
   rank 1 treats from n_bytes//nb_nodes to 2*(n_bytes//size) - 1 + (max_len_pattern - 1)
   .
   rank i treat from i*(n_bytes//nb_nodes) to (i+1)*(n_bytes//size) - 1 + (max_len_pattern - 1)
   .
   rank (nb_nodes-1) treat from (nb_nodes-1)*(n_bytes//nb_nodes) to END
*/

/* rank 0 play the role of divider */
int part_bytes; // the number of bytes of the process part textfile
MPI_Request requests[nb_nodes-1];
MPI_Status statutes[nb_nodes-1];
if (rank == 0) {
  /* reading input file */
  buf = read_input_file( filename, &n_bytes ) ;
  if ( buf == NULL )
  {
      return 1 ;
  }

  int start = 0; // start index of process
  int end = n_bytes/nb_nodes - 1 + (max_len_pattern - 1); // end index of process
  #if APM_DEBUG
      printf( "MPI rank 0 will treat from bytes %d to %d\n", start, end);
  #endif
```



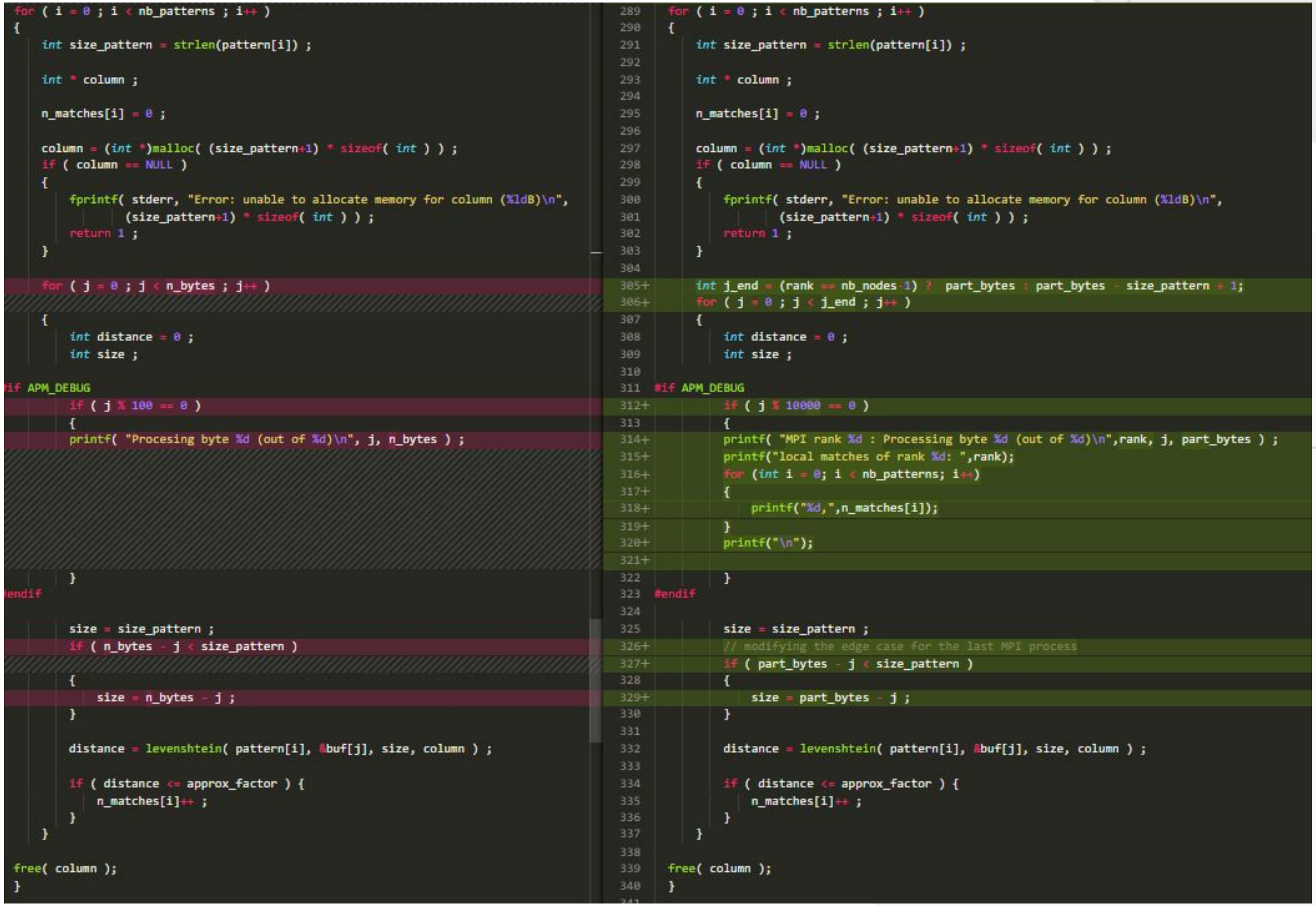
```
  for (int i = 1; i < nb_nodes; i++) {
    /* Index and process part bytes */
    start += (n_bytes/nb_nodes);
    end += (n_bytes/nb_nodes);
    if (i == nb_nodes - 1 || end > n_bytes) {
        end = n_bytes;
    }
    part_bytes = end - start + 1 ;
    #if APM_DEBUG
        printf("MPI rank %d will treat from bytes %d to %d\n",i,start,end);
    #endif
    /* Sending to each process other than 0*/
    /* the part_bytes so they how much memory to allocate */
    MPI_Send(&part_bytes,1,MPI_INTEGER,i,0,MPI_COMM_WORLD);
    #if APM_DEBUG
        printf("Rank 0 sended part_bytes : %d to rank %d\n",part_bytes,i);
    #endif
    MPI_Send(&buf[start],part_bytes,MPI_BYTE,i,1,MPI_COMM_WORLD);
    #if APM_DEBUG
        printf("Rank 0 sended a part_buffer to rank %d\n",i);
    #endif
    /* the start & end index of their part */
  }
```



```
  // Reset part_bytes for process 0 :
  part_bytes = n_bytes/nb_nodes - 1 + max_len_pattern - 1;
} else {
  // other process receive :
  // first : part_bytes :
  MPI_Recv(&part_bytes,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,&status);
  // so they know how much to allocate
  buf = (char *) malloc((part_bytes+1)*sizeof(char));
  if ( buf == NULL )
  {
    fprintf( stderr, "Unable to allocate %ld byte(s) for buf array\n",part_bytes);
    return -1;
  }
  // secondly : part textfile :
  MPI_Recv(buf,part_bytes,MPI_BYTE,0,1,MPI_COMM_WORLD,&status);
}
```

MPI

Boucle principale

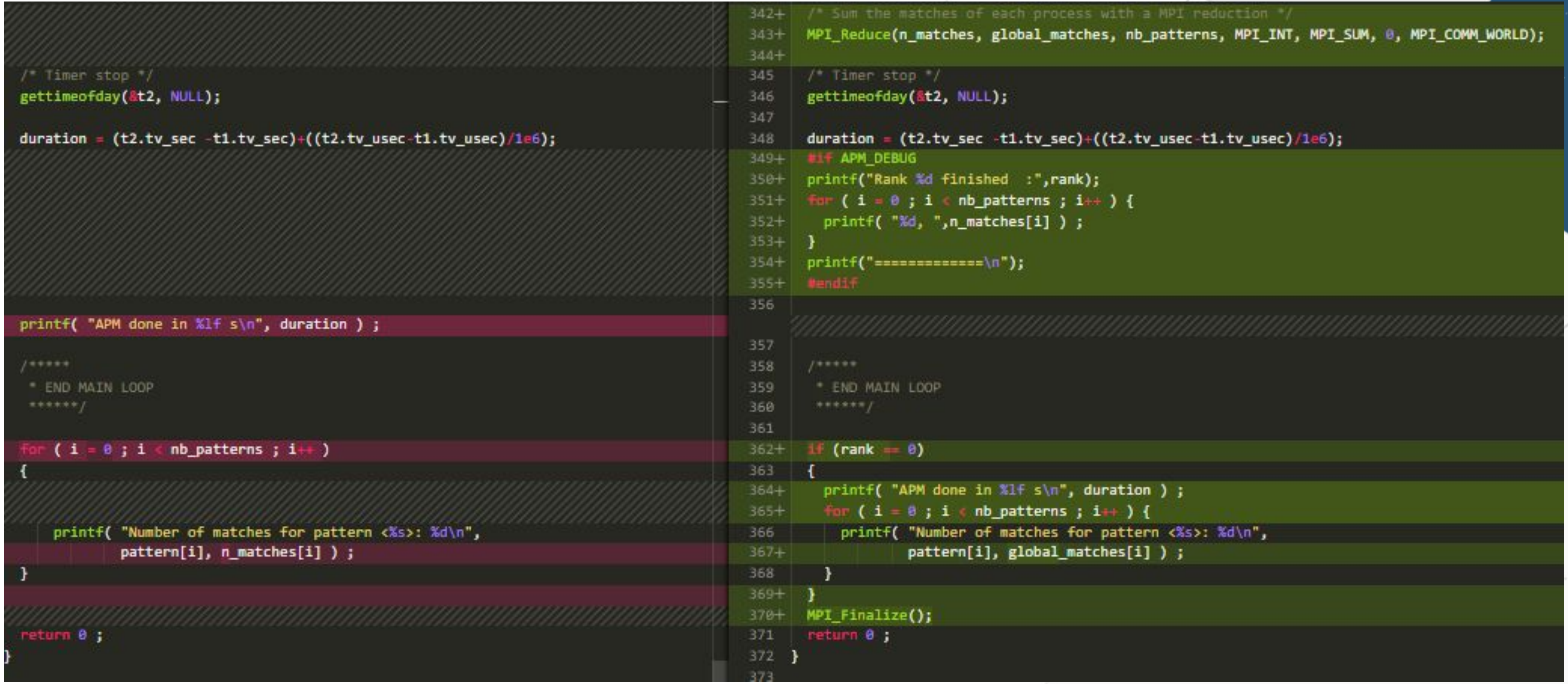


```
for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
    if ( column == NULL )
    {
        fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
                (size_pattern+1) * sizeof( int ) ) ;
        return 1 ;
    }

    for ( j = 0 ; j < n_bytes ; j++ )
    {
        int distance = 0 ;
        int size ;

if APM_DEBUG
        if ( j % 100 == 0 )
        {
        printf( "Procesing byte %d (out of %d)\n", j, n_bytes ) ;
        }
endif

        size = size_pattern ;
        if ( n_bytes - j < size_pattern )
        {
            size = n_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            n_matches[i]++ ;
        }
    }

free( column );
}
```

```
for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
    if ( column == NULL )
    {
        fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
                (size_pattern+1) * sizeof( int ) ) ;
        return 1 ;
    }

    int j_end = (rank == nb_nodes-1) ?  part_bytes : part_bytes - size_pattern + 1;
    for ( j = 0 ; j < j_end ; j++ )
    {
        int distance = 0 ;
        int size ;

#if APM_DEBUG
        if ( j % 10000 == 0 )
        {
        printf( "MPI rank %d : Processing byte %d (out of %d)\n",rank, j, part_bytes ) ;
        printf("local matches of rank %d: ",rank);
        for (int i = 0; i < nb_patterns; i++)
        {
            printf("%d,",n_matches[i]);
        }
        printf("\n");

        }
#endif

        size = size_pattern ;
        // modifying the edge case for the last MPI process
        if ( part_bytes - j < size_pattern )
        {
            size = part_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            n_matches[i]++ ;
        }
    }

free( column );
}
```

MPI

Main

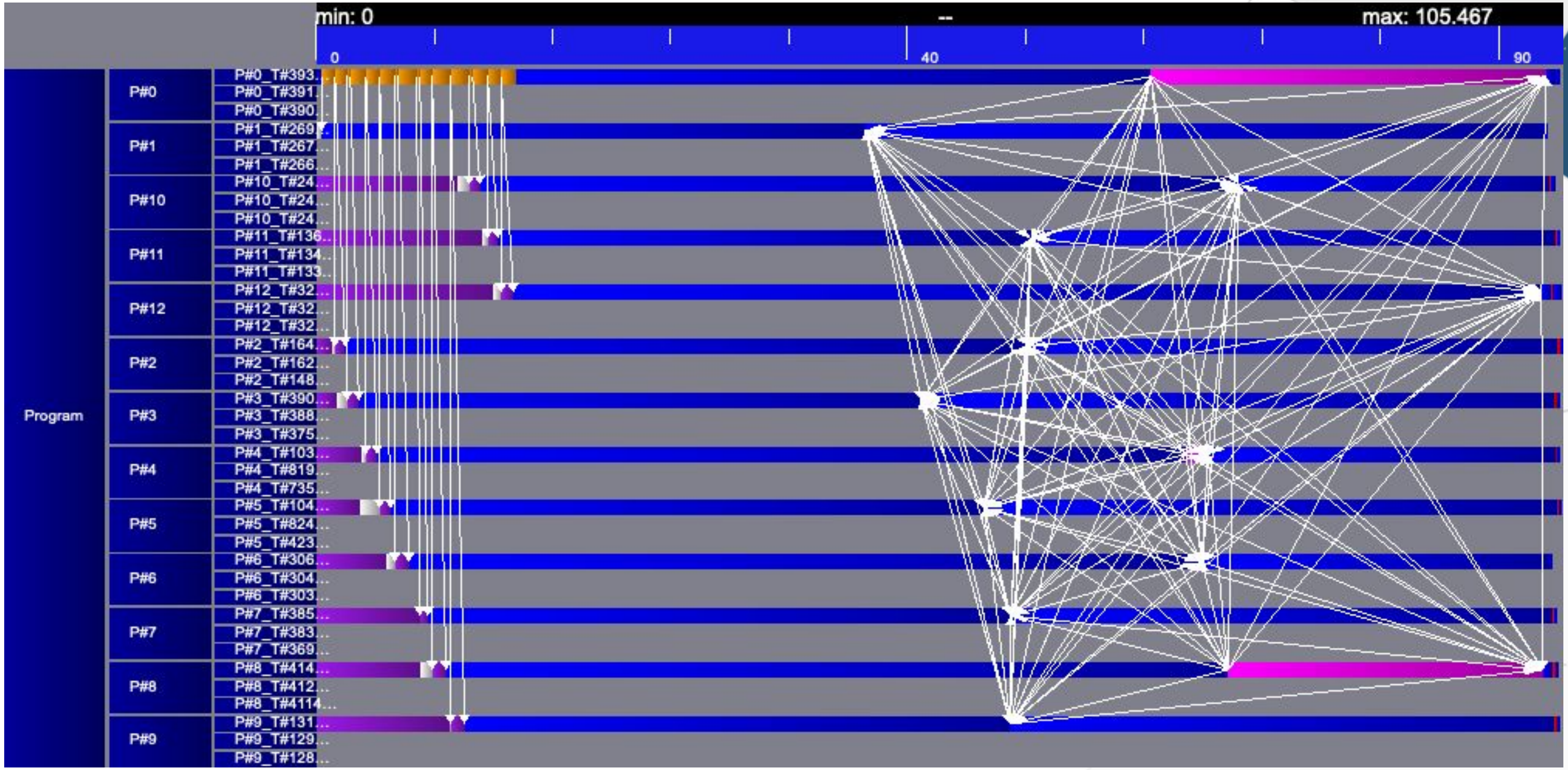


```
/* Timer stop */
gettimeofday(&t2, NULL);

duration = (t2.tv_sec -t1.tv_sec)+((t2.tv_usec-t1.tv_usec)/1e6);

printf( "APM done in %lf s\n", duration ) ;

/*****
 * END MAIN LOOP
 ******/

for ( i = 0 ; i < nb_patterns ; i++ )
{
    printf( "Number of matches for pattern <%s>: %d\n",
            pattern[i], n_matches[i] ) ;
}

return 0 ;
}
```

```
+ /* Sum the matches of each process with a MPI reduction */
+ MPI_Reduce(n_matches, global_matches, nb_patterns, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
+
  /* Timer stop */
  gettimeofday(&t2, NULL);

  duration = (t2.tv_sec -t1.tv_sec)+((t2.tv_usec-t1.tv_usec)/1e6);
+ #if APM_DEBUG
+ printf("Rank %d finished  :",rank);
+ for ( i = 0 ; i < nb_patterns ; i++ ) {
+   printf( "%d, ",n_matches[i] ) ;
+ }
+ printf("=============\n");
+ #endif


  /*****
   * END MAIN LOOP
   ******/

+ if (rank == 0)
  {
+   printf( "APM done in %lf s\n", duration ) ;
+   for ( i = 0 ; i < nb_patterns ; i++ ) {
      printf( "Number of matches for pattern <%s>: %d\n",
+             pattern[i], global_matches[i] ) ;
    }
+ }
+ MPI_Finalize();
  return 0 ;
}
```

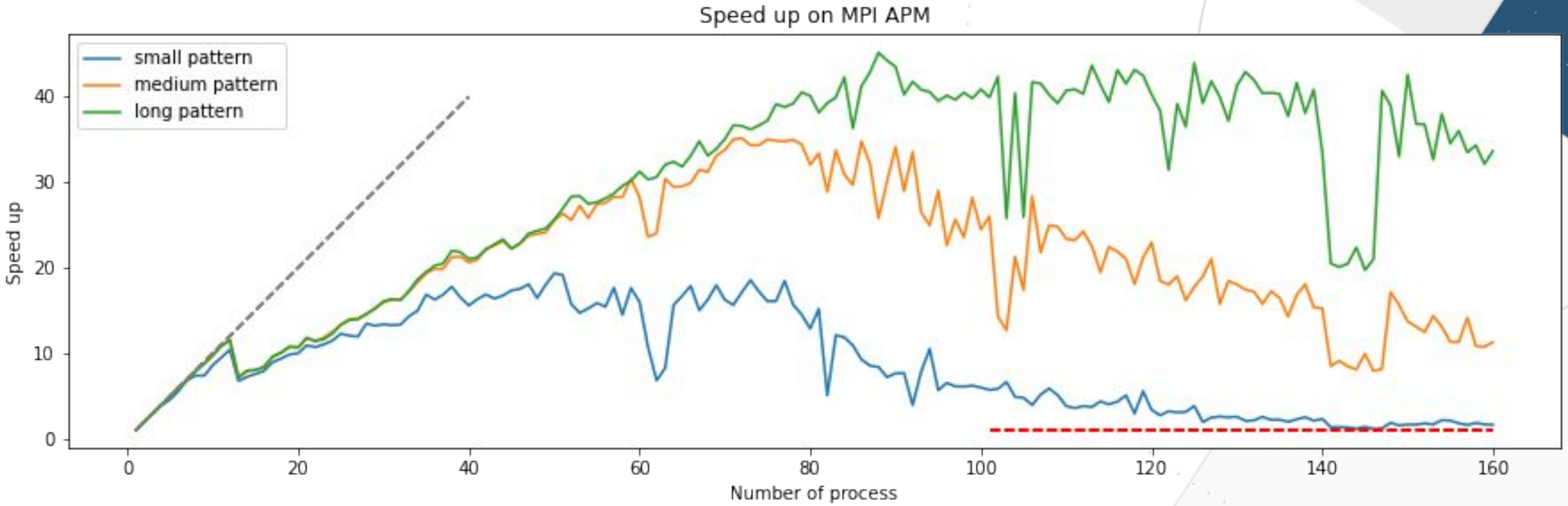
MPI

Trace


min: 0
--
max: 105.467
0
40
90
Program
P#0
P#0_T#393...
P#0_T#391...
P#0_T#390...
P#1
P#1_T#269...
P#1_T#267...
P#1_T#266...
P#10
P#10_T#24...
P#10_T#24...
P#10_T#24...
P#11
P#11_T#136...
P#11_T#134...
P#11_T#133...
P#12
P#12_T#32...
P#12_T#32...
P#12_T#32...
P#2
P#2_T#164...
P#2_T#162...
P#2_T#148...
P#3
P#3_T#390...
P#3_T#388...
P#3_T#375...
P#4
P#4_T#103...
P#4_T#819...
P#4_T#735...
P#5
P#5_T#104...
P#5_T#824...
P#5_T#423...
P#6
P#6_T#306...
P#6_T#304...
P#6_T#303...
P#7
P#7_T#385...
P#7_T#383...
P#7_T#369...
P#8
P#8_T#414...
P#8_T#412...
P#8_T#4114...
P#9
P#9_T#131...
P#9_T#129...
P#9_T#128...

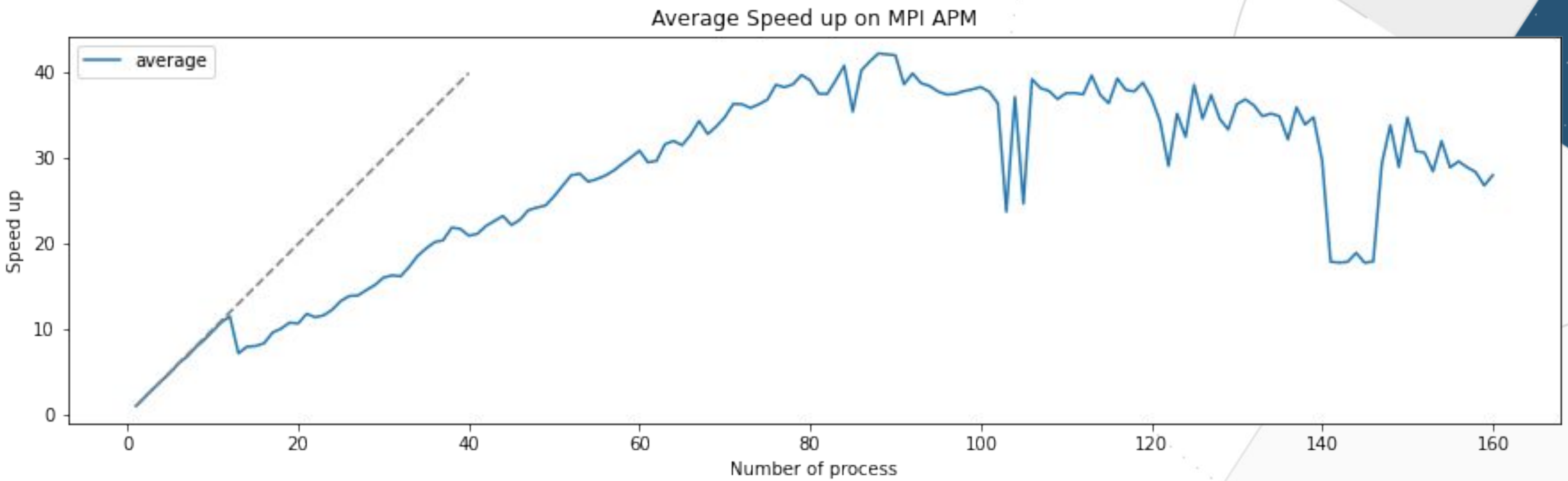
MPI

SpeedUp


Speed up on MPI APM
small pattern
medium pattern
long pattern
Speed up
0
10
20
30
40
0
20
40
60
80
100
120
140
160
Number of process

MPI

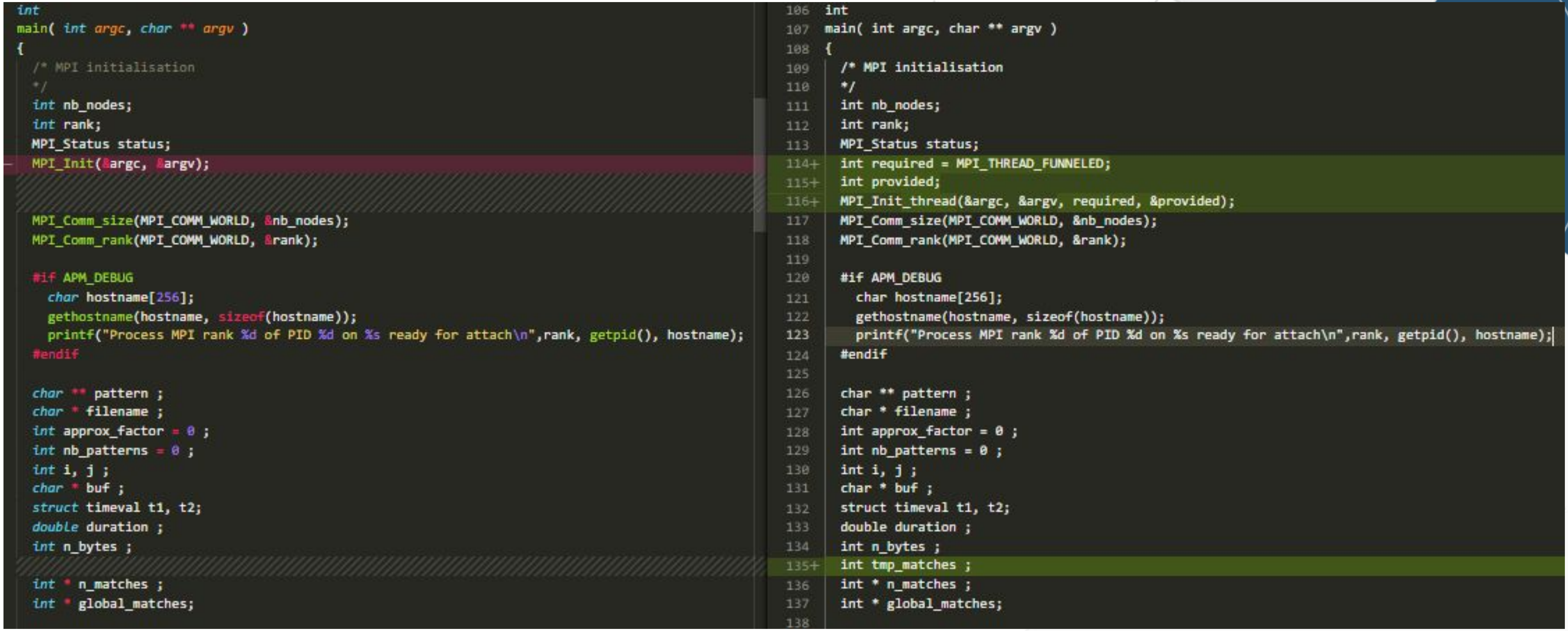
SpeedUp


Average Speed up on MPI APM
average
Speed up
0
10
20
30
40
Number of process
0
20
40
60
80
100
120
140
160

OpenMP+MPI

OpenMP+MPI

Main

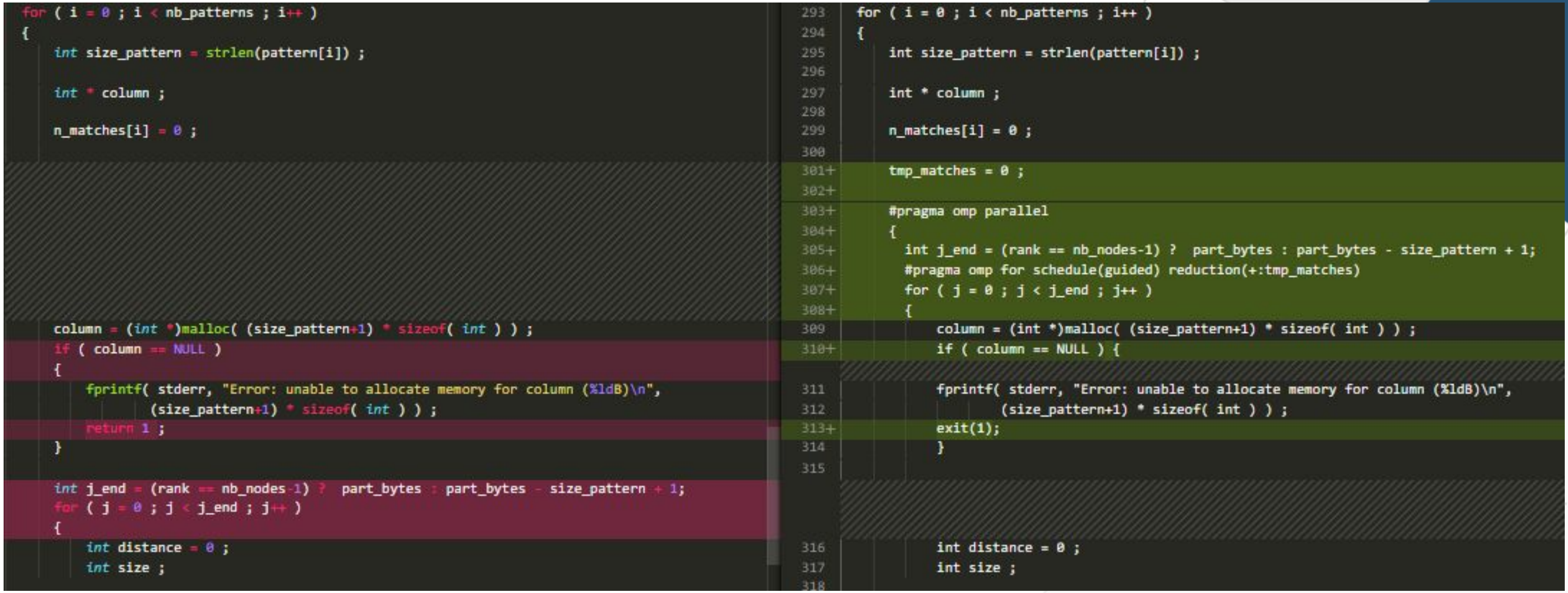


```c
int
main( int argc, char ** argv )
{
  /* MPI initialisation
  */
  int nb_nodes;
  int rank;
  MPI_Status status;
- MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &nb_nodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  #if APM_DEBUG
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    printf("Process MPI rank %d of PID %d on %s ready for attach\n",rank, getpid(), hostname);
  #endif

  char ** pattern ;
  char * filename ;
  int approx_factor = 0 ;
  int nb_patterns = 0 ;
  int i, j ;
  char * buf ;
  struct timeval t1, t2;
  double duration ;
  int n_bytes ;

  int * n_matches ;
  int * global_matches;
```

```c
int
main( int argc, char ** argv )
{
  /* MPI initialisation
  */
  int nb_nodes;
  int rank;
  MPI_Status status;
+ int required = MPI_THREAD_FUNNELED;
+ int provided;
+ MPI_Init_thread(&argc, &argv, required, &provided);
  MPI_Comm_size(MPI_COMM_WORLD, &nb_nodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  #if APM_DEBUG
    char hostname[256];
    gethostname(hostname, sizeof(hostname));
    printf("Process MPI rank %d of PID %d on %s ready for attach\n",rank, getpid(), hostname);
  #endif

  char ** pattern ;
  char * filename ;
  int approx_factor = 0 ;
  int nb_patterns = 0 ;
  int i, j ;
  char * buf ;
  struct timeval t1, t2;
  double duration ;
  int n_bytes ;
+ int tmp_matches ;
  int * n_matches ;
  int * global_matches;

```

OpenMP+MPI

Main

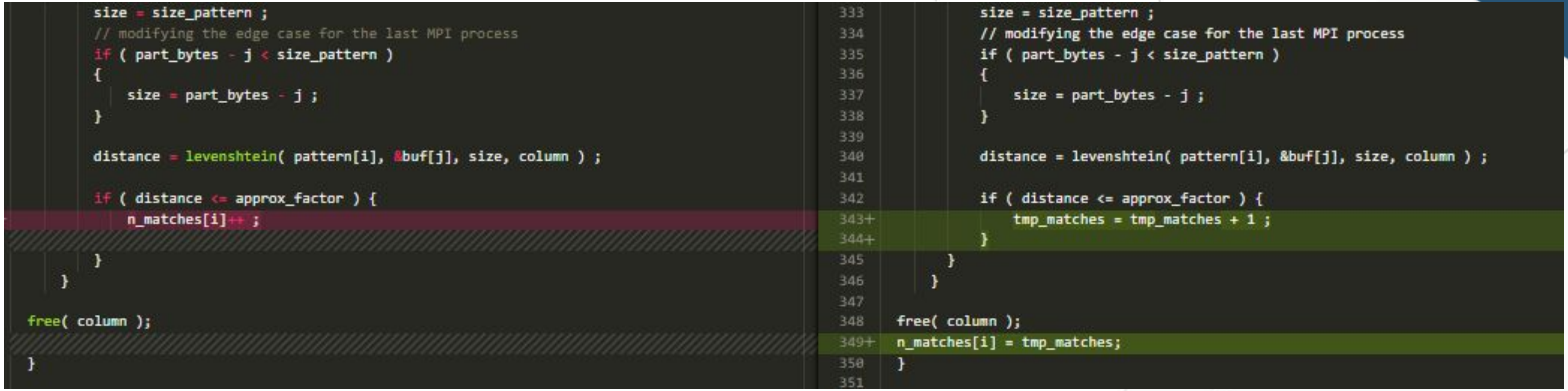


```
for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
    if ( column == NULL )
    {
        fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
                (size_pattern+1) * sizeof( int ) ) ;
        return 1 ;
    }

    int j_end = (rank == nb_nodes-1) ? part_bytes : part_bytes - size_pattern + 1;
    for ( j = 0 ; j < j_end ; j++ )
    {
        int distance = 0 ;
        int size ;
```

```
for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    tmp_matches = 0 ;

    #pragma omp parallel
    {
      int j_end = (rank == nb_nodes-1) ?  part_bytes : part_bytes - size_pattern + 1;
      #pragma omp for schedule(guided) reduction(+:tmp_matches)
      for ( j = 0 ; j < j_end ; j++ )
      {
        column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
        if ( column == NULL ) {
          fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
                (size_pattern+1) * sizeof( int ) ) ;
          exit(1);
          }

          int distance = 0 ;
          int size ;
```

OpenMP+MPI

Main



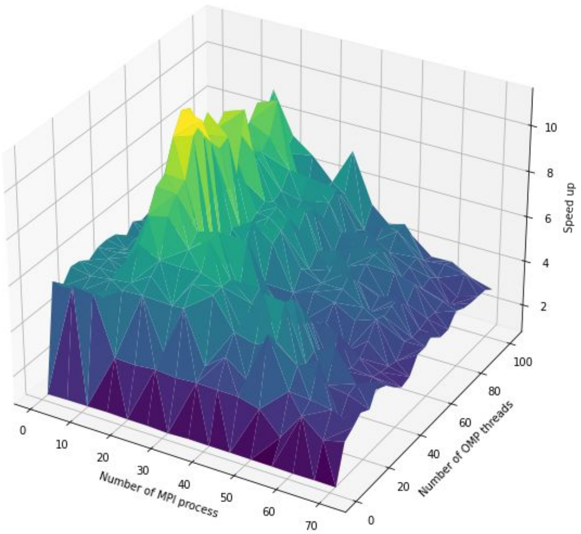
```c
        size = size_pattern ;
        // modifying the edge case for the last MPI process
        if ( part_bytes - j < size_pattern )
        {
            size = part_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            n_matches[i]++ ;

        }
    }

free( column );

}
```

```c
        size = size_pattern ;
        // modifying the edge case for the last MPI process
        if ( part_bytes - j < size_pattern )
        {
            size = part_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            tmp_matches = tmp_matches + 1 ;
        }
        }
    }

free( column );
n_matches[i] = tmp_matches;
}
```

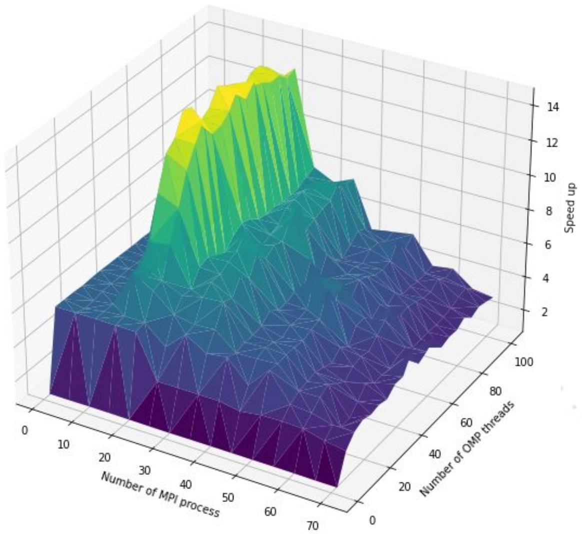
OpenMP+MPI

Speed up

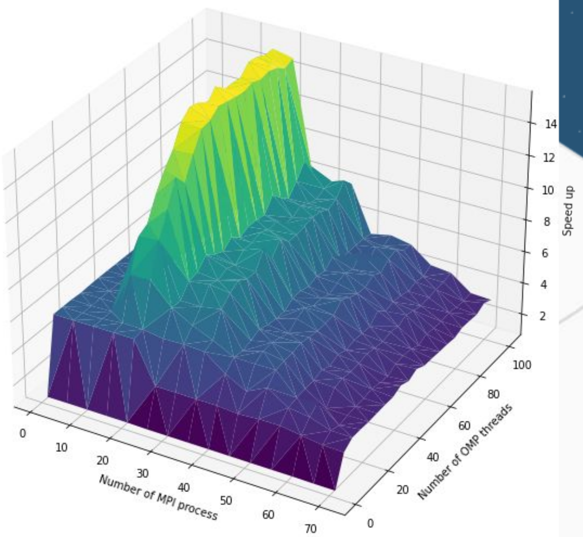
MPI and OMP APM with small pattern



MPI and OMP APM with medium pattern



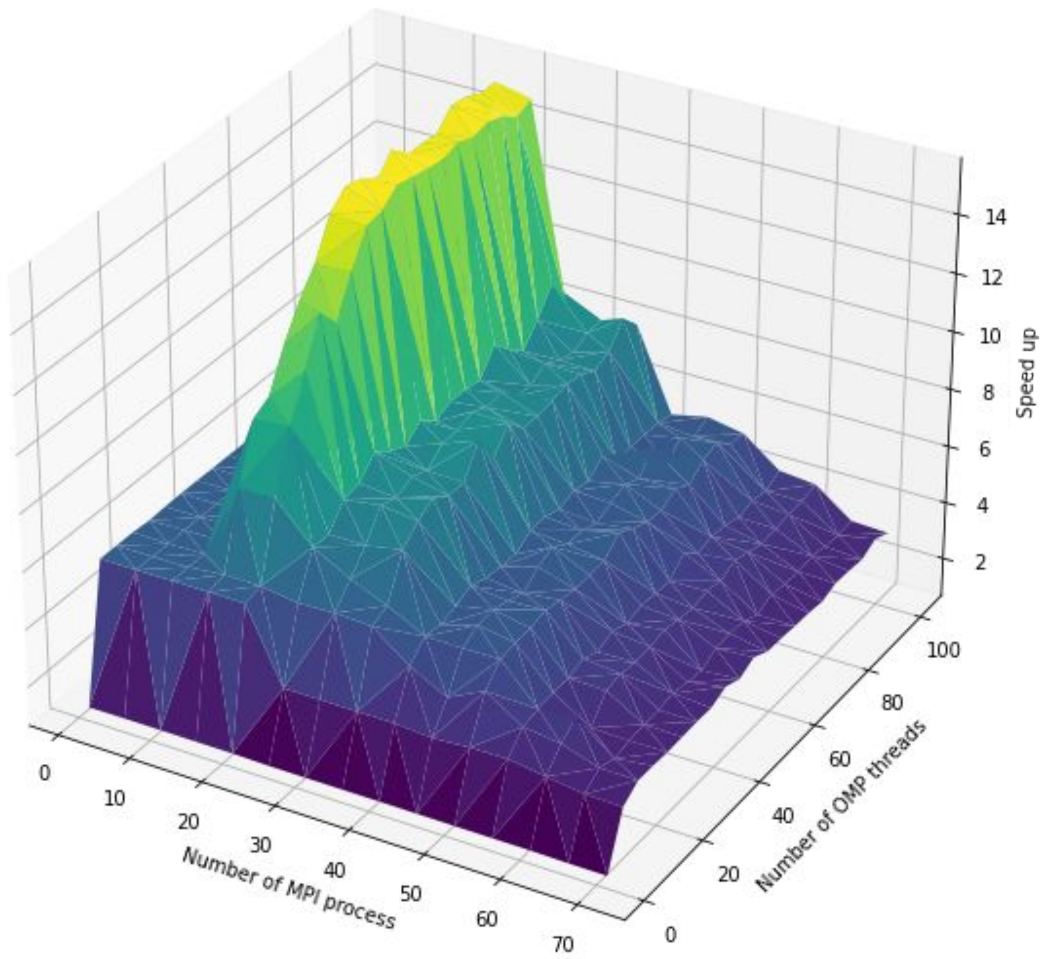
MPI and OMP APM with large pattern

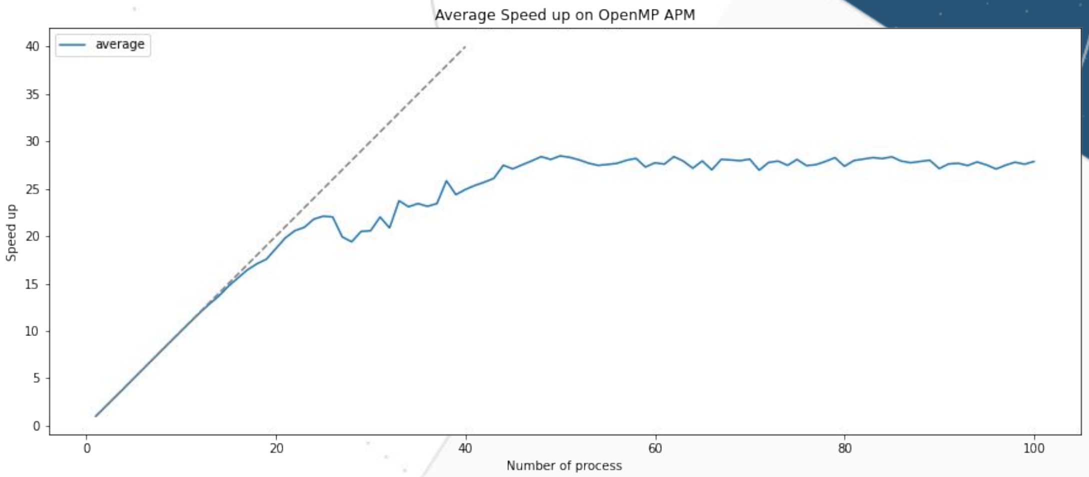


OpenMP+MPI

Speed up

MPI and OMP APM with large pattern





CUDA

CUDA

Idée générale

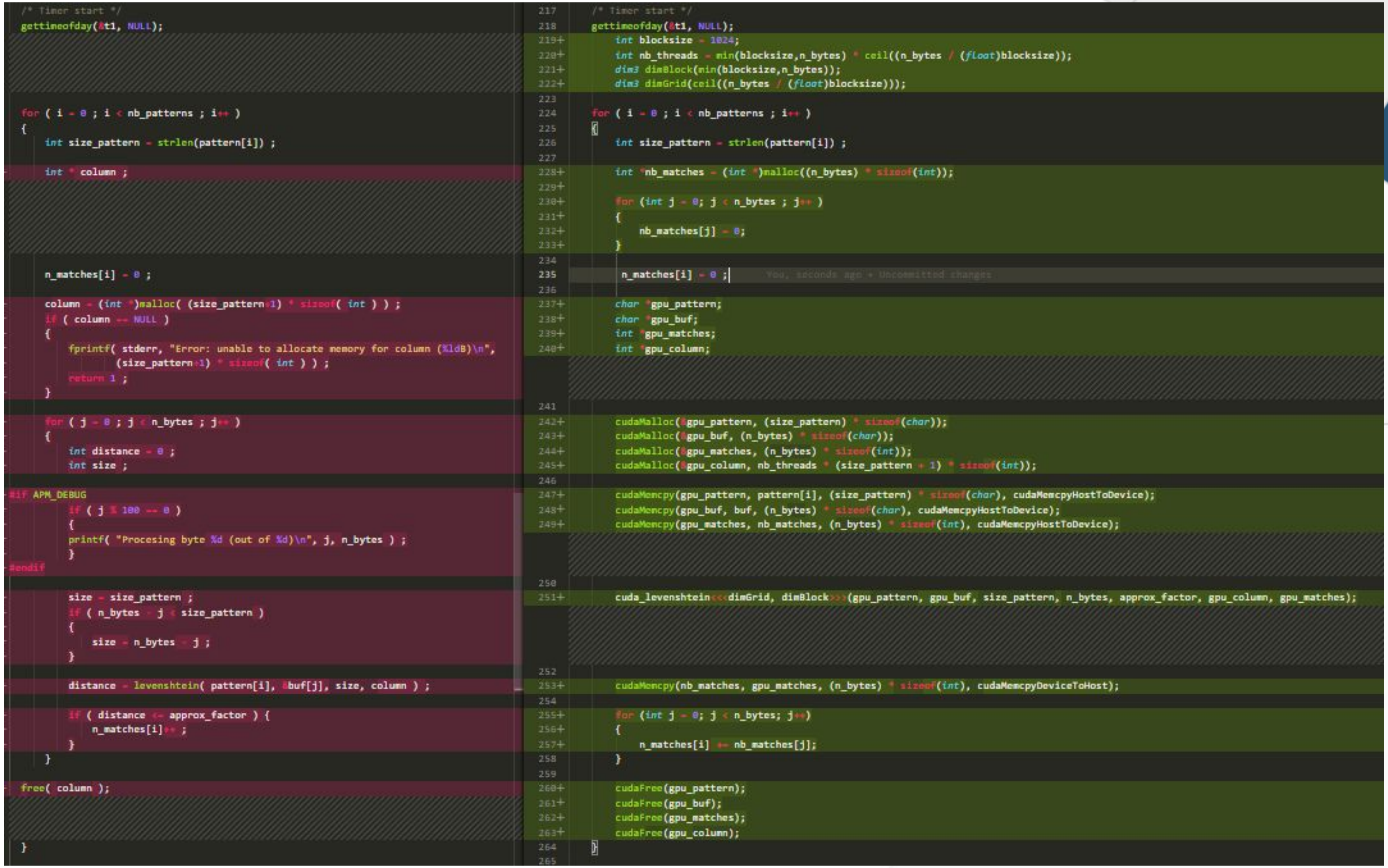
ADN:
ACGT

Principe:
Calcul départ de A sur un GPU
Calcul départ de C sur un GPU
…

Problème:
Chaque GPU doit avoir en mémoire le pattern, la séquence d'ADN, … (les paramètres en général) et un paramètre sur lequel stocker la distance à récupérer.

CUDA

Main

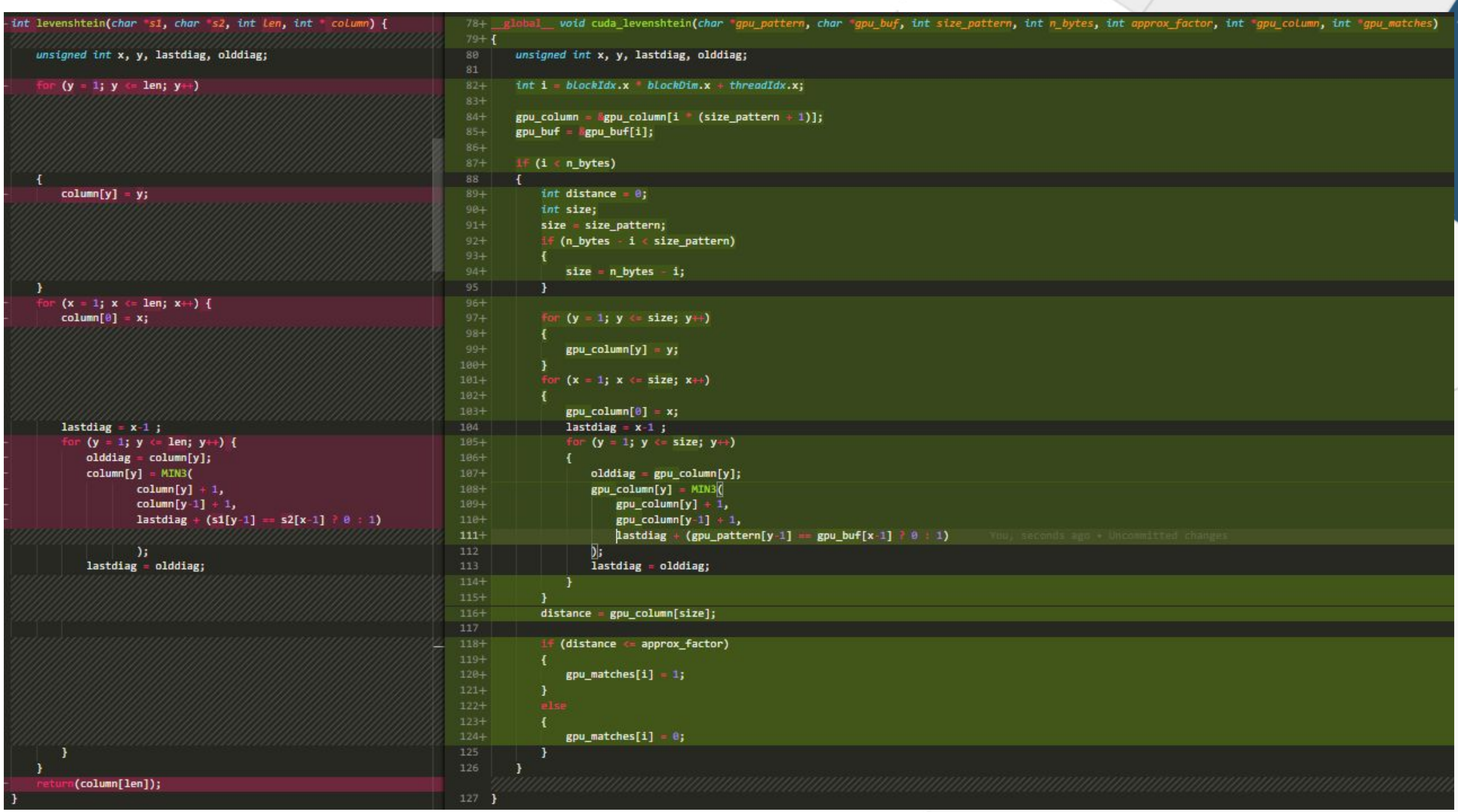


```c
/* Timer start */
gettimeofday(&t1, NULL);

for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int * column ;

    n_matches[i] = 0 ;

    column = (int *)malloc( (size_pattern+1) * sizeof( int ) ) ;
    if ( column == NULL )
    {
        fprintf( stderr, "Error: unable to allocate memory for column (%ldB)\n",
                (size_pattern+1) * sizeof( int ) ) ;
        return 1 ;
    }

    for ( j = 0 ; j < n_bytes ; j++ )
    {
        int distance = 0 ;
        int size ;

#if APM_DEBUG
        if ( j % 100 == 0 )
        {
        printf( "Procesing byte %d (out of %d)\n", j, n_bytes ) ;
        }
#endif

        size = size_pattern ;
        if ( n_bytes - j < size_pattern )
        {
            size = n_bytes - j ;
        }

        distance = levenshtein( pattern[i], &buf[j], size, column ) ;

        if ( distance <= approx_factor ) {
            n_matches[i]++ ;
        }
    }

free( column );

}
```

```c
/* Timer start */
gettimeofday(&t1, NULL);
    int blocksize = 1024;
    int nb_threads = min(blocksize,n_bytes) * ceil((n_bytes / (float)blocksize));
    dim3 dimBlock(min(blocksize,n_bytes));
    dim3 dimGrid(ceil((n_bytes / (float)blocksize)));

for ( i = 0 ; i < nb_patterns ; i++ )
{
    int size_pattern = strlen(pattern[i]) ;

    int *nb_matches = (int *)malloc((n_bytes) * sizeof(int));

    for (int j = 0; j < n_bytes ; j++ )
    {
        nb_matches[j] = 0;
    }

     n_matches[i] = 0 ;        You, seconds ago • Uncommitted changes

    char *gpu_pattern;
    char *gpu_buf;
    int *gpu_matches;
    int *gpu_column;

    cudaMalloc(&gpu_pattern, (size_pattern) * sizeof(char));
    cudaMalloc(&gpu_buf, (n_bytes) * sizeof(char));
    cudaMalloc(&gpu_matches, (n_bytes) * sizeof(int));
    cudaMalloc(&gpu_column, nb_threads * (size_pattern + 1) * sizeof(int));

    cudaMemcpy(gpu_pattern, pattern[i], (size_pattern) * sizeof(char), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_buf, buf, (n_bytes) * sizeof(char), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_matches, nb_matches, (n_bytes) * sizeof(int), cudaMemcpyHostToDevice);

    cuda_levenshtein<<<dimGrid, dimBlock>>>(gpu_pattern, gpu_buf, size_pattern, n_bytes, approx_factor, gpu_column, gpu_matches);

    cudaMemcpy(nb_matches, gpu_matches, (n_bytes) * sizeof(int), cudaMemcpyDeviceToHost);

    for (int j = 0; j < n_bytes; j++)
    {
        n_matches[i] += nb_matches[j];
    }

    cudaFree(gpu_pattern);
    cudaFree(gpu_buf);
    cudaFree(gpu_matches);
    cudaFree(gpu_column);
}
```

CUDA

Levenshtein

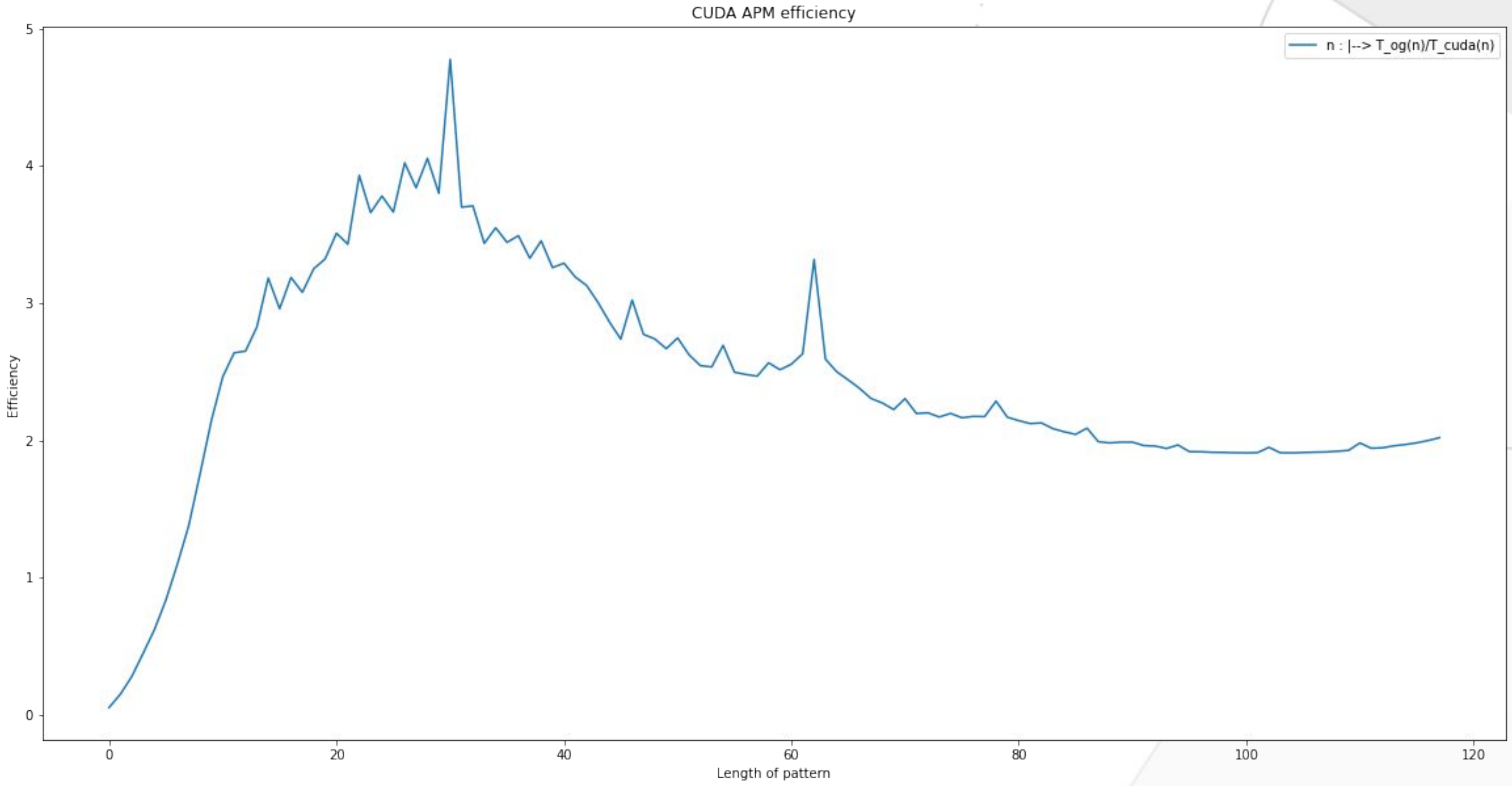


```
int levenshtein(char *s1, char *s2, int len, int * column) {

    unsigned int x, y, lastdiag, olddiag;

    for (y = 1; y <= len; y++)
    {
        column[y] = y;
    }
    for (x = 1; x <= len; x++) {
        column[0] = x;
        lastdiag = x-1 ;
        for (y = 1; y <= len; y++) {
            olddiag = column[y];
            column[y] = MIN3(
                    column[y] + 1,
                    column[y-1] + 1,
                    lastdiag + (s1[y-1] == s2[x-1] ? 0 : 1)
                    );
            lastdiag = olddiag;
        }
    }
    return(column[len]);
}
```

```
__global__ void cuda_levenshtein(char *gpu_pattern, char *gpu_buf, int size_pattern, int n_bytes, int approx_factor, int *gpu_column, int *gpu_matches)
{
    unsigned int x, y, lastdiag, olddiag;

    int i = blockIdx.x * blockDim.x + threadIdx.x;

    gpu_column = &gpu_column[i * (size_pattern + 1)];
    gpu_buf = &gpu_buf[i];

    if (i < n_bytes)
    {
        int distance = 0;
        int size;
        size = size_pattern;
        if (n_bytes - i < size_pattern)
        {
            size = n_bytes - i;
        }

        for (y = 1; y <= size; y++)
        {
            gpu_column[y] = y;
        }
        for (x = 1; x <= size; x++)
        {
            gpu_column[0] = x;
            lastdiag = x-1 ;
            for (y = 1; y <= size; y++)
            {
                olddiag = gpu_column[y];
                gpu_column[y] = MIN3(
                    gpu_column[y] + 1,
                    gpu_column[y-1] + 1,
                    lastdiag + (gpu_pattern[y-1] == gpu_buf[x-1] ? 0 : 1)        You, seconds ago • Uncommitted changes
                );
                lastdiag = olddiag;
            }
        }
        distance = gpu_column[size];

        if (distance <= approx_factor)
        {
            gpu_matches[i] = 1;
        }
        else
        {
            gpu_matches[i] = 0;
        }
    }
}
```

CUDA

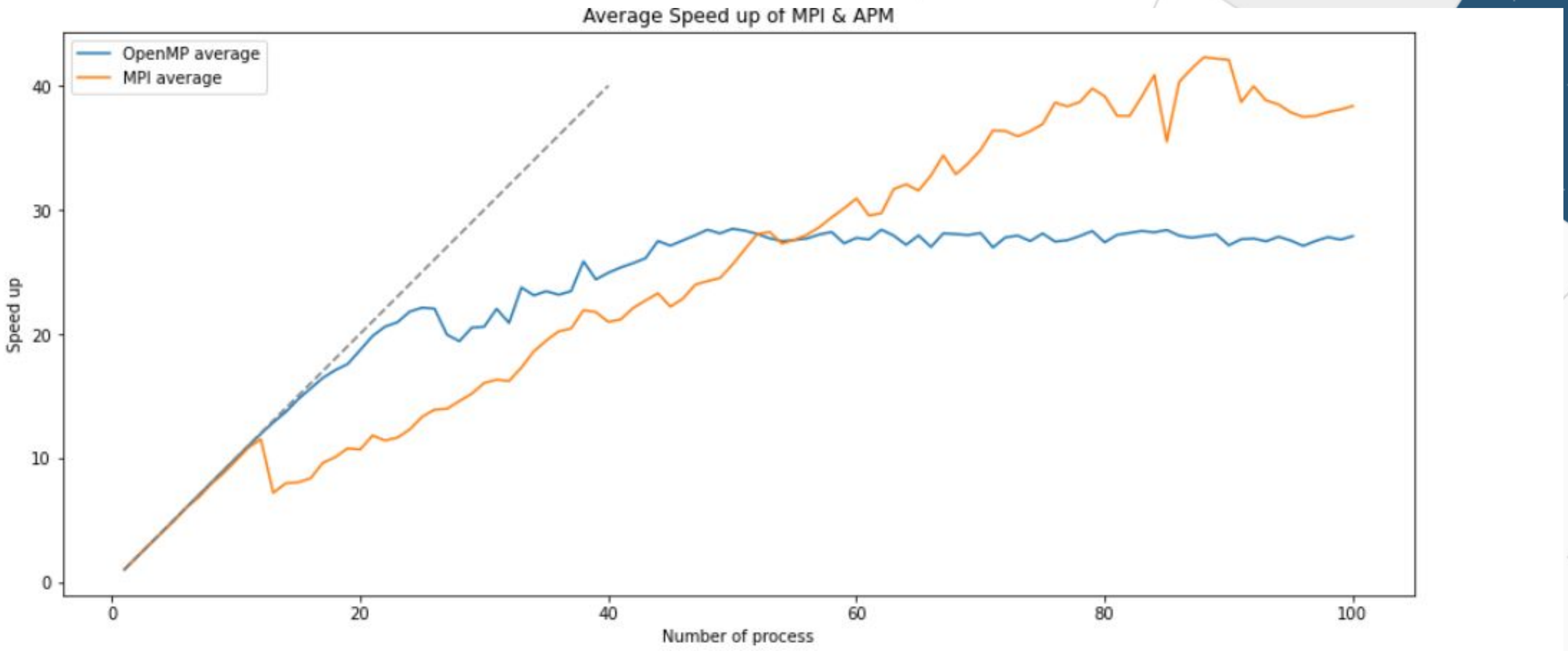
Efficiency



Comparaison

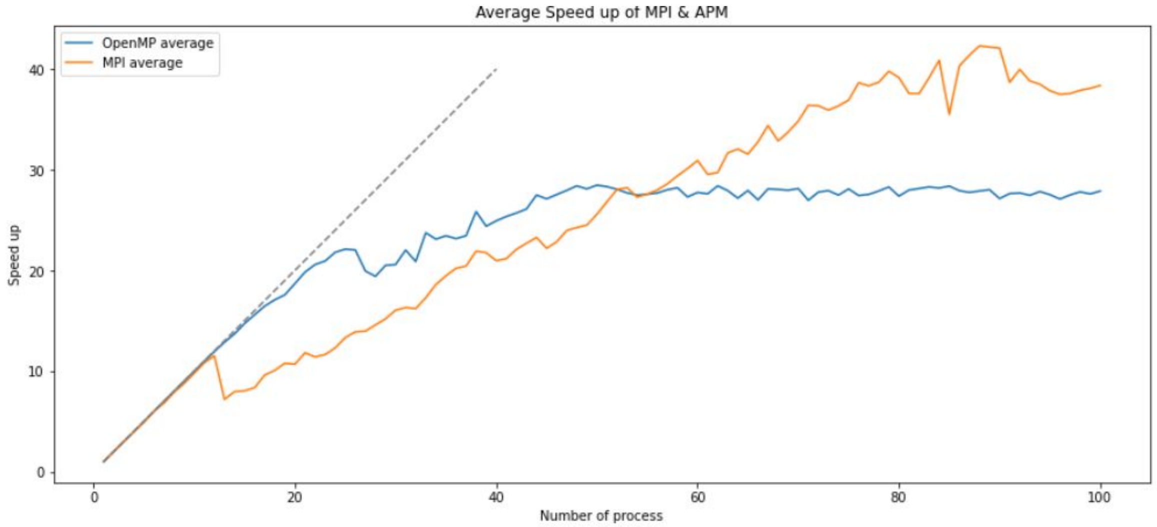
Comparaison

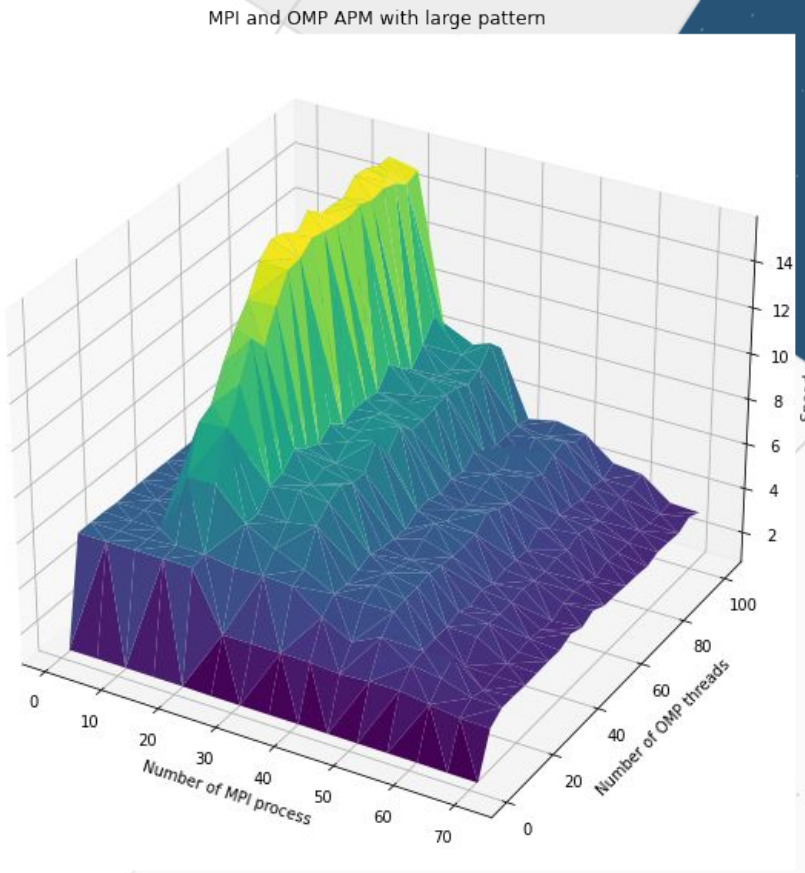
SpeedUp OpenMp et MPI



Comparaison

SpeedUp OpenMp et MPI


Average Speed up of MPI & APM
OpenMP average
MPI average
Speed up
Number of process


MPI and OMP APM with large pattern
Speed up
Number of MPI process
Number of OMP threads