PACO Laboratory Assignment

Lab 5: Parallel Data Decomposition Implementation and

Analysis

|  | Number of threads (elapsed | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| Version | 1 | 4 | 8 | 12 | 16 | 20 |
| 1D Block Geometric Data Decomposition by columns | 2.96 | 2.09 | 1.54 | 1.18 | 1.01 | 0.87 |
| 1D Block-Cyclic Geometric Data Decomposition by columns | 2.91 | 0.74 | 0.414 | 0.3 | 0.245 | 0.21 |
| 1D Cyclic Geometric Data Decomposition by rows | 2.885 | 0.8 | 0.42 | 0.286 | 0.234 | 0.18 |
|  | Number of threads (L2 Cache Misses per thread) | | | | | |
| 1D Block Geometric Data Decomposition by columns | 1643594 | 1977117 | 2344291 | 2752432 | 2695098 | 3004044 |
| 1D Block-Cyclic Geometric Data Decomposition by columns | 1642045 | 7373545 | 13838017 | 21116201 | 24670958 | 24666454 |
| 1D Cyclic Geometric Data Decomposition by rows | 1641772 | 1648665 | 1655372 | 1666835 | 1670766 | 1685594 |

Analisis:

1D Block Geometric Data Decomposition by columns

En aquest primer codi calculem la imatge per columnes. Per cada thread li assignem un numero de columnes depenent del numero total de columnes.   
El problema de aquest codi és que si el numero de columnes no es divisible per el numero de threads, hi quedaran columnes sense assignar. Per arreglar això hem afegit al codi el següent:

if((reminder-my\_thread)>0) {

int px = (COLS-1) - my\_thread;

M[py][px] = pixel\_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, py, scale\_real, scale\_imag, maxiter);

if (output2histogram)

#pragma omp atomic

histogram[M[py][px] - 1]++;

if (output2display) {

/\* Scale color and display point \*/

#pragma omp critical

{

long color = (long)((M[py][px] - 1) \* scale\_color) + min\_color;

if (setup\_return == EXIT\_SUCCESS) {

XSetForeground(display, gc, color);

XDrawPoint(display, win, gc, px, py);

}

}

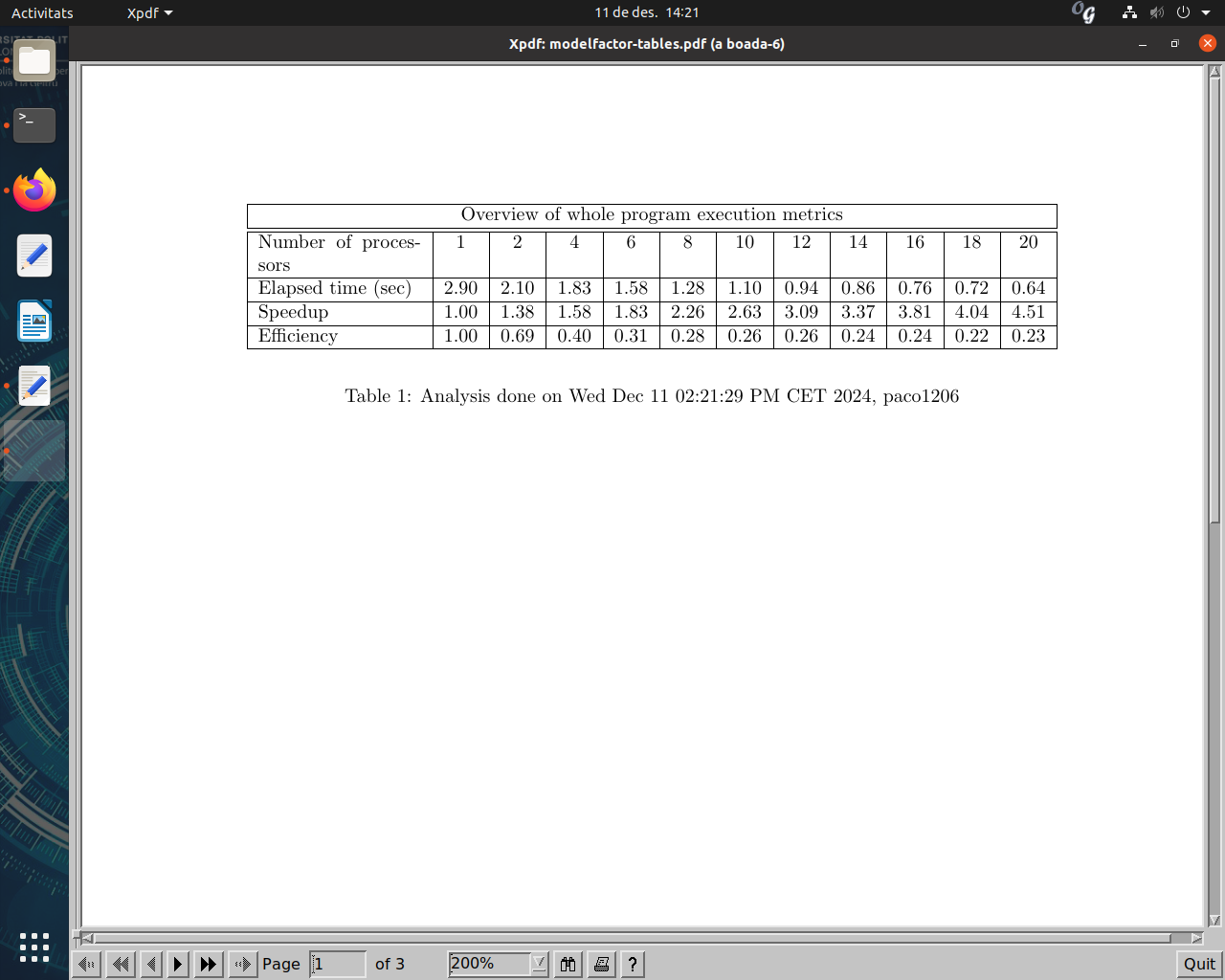
}

}

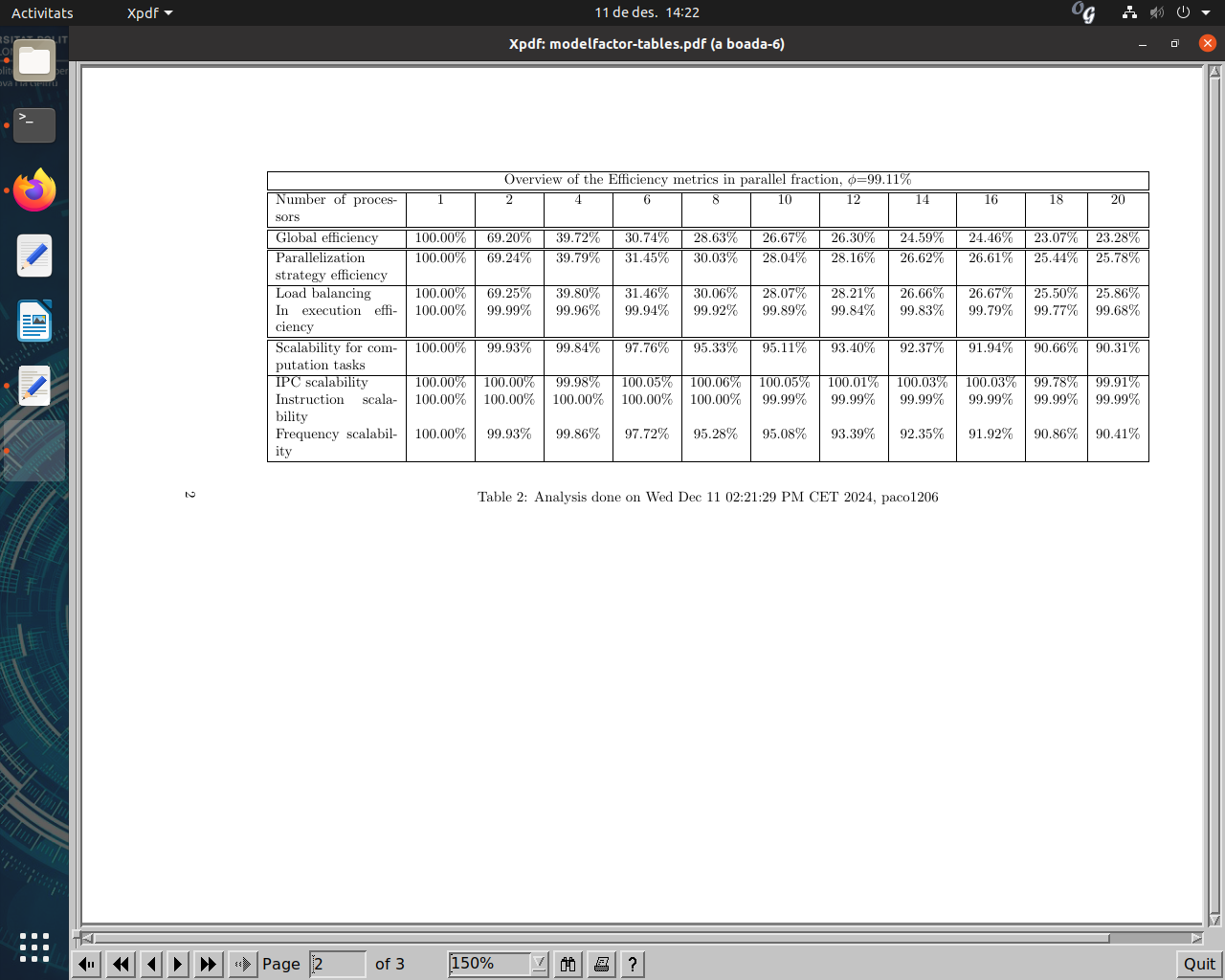


Aquest codi reparteix les columnes restants s’assignen a les primeres columnes.

**Modelfactor analysis:**

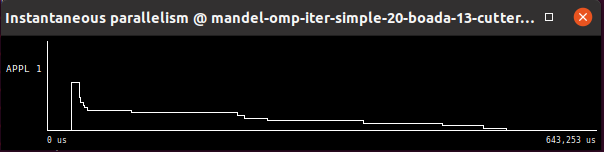


En aquesta primera taula podem observar que, a mesura que afegim processadors, el temps d'execució disminueix, cosa que confirma que el paral·lelisme és efectiu. El speedup també augmenta, tot i que no de forma lineal, ja que es veu afectat pels costos de coordinació i distribució del treball. En canvi, l'eficiència baixa considerablement amb l'increment del nombre de processadors, fet que indica que alguns threads fan més treball que altres. Aquesta baixada d'eficiència es deu a les columnes restants que queden sense repartir equitativament entre els threads quan el nombre total de columnes no és divisible de manera exacta pel nombre de processadors.



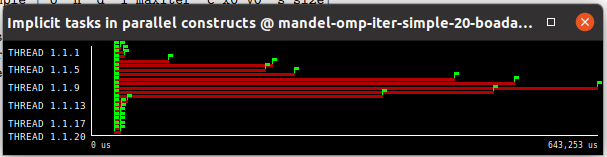
Aquesta segona taula ens torna a mostrar que la càrrega de treball és bastant baixa a mesura que s’afegeixen més threads. Això pot ser degut al fet que encara que repartim el treball per columnes, poden haver-hi columnes més costoses que altres fent que alguns threads tinguin més treball, a més si aquests threads són els que els hi assignen les columnes sobrants, la càrrega de treball serà pitjor encara. A més l’estratègia de paral·lelització tampoc és bona, això pot ser pel mètode de calcular la imatge per columnes de la manera que ho fem, ja que columnes podrien dependre entre si o si assignem les columnes restants a threads que ja tenir molta càrrega de treball.

**Paraver Analysis:**

****

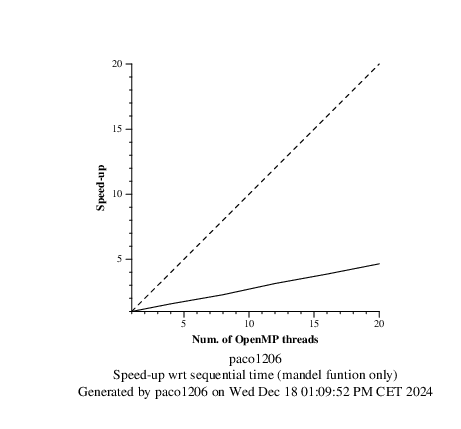
Aquesta primera gràfica ens mostra el paral·lelisme a través de l’execució del codi.

Es pot comprovar que al princpi de l’execució el paralelisme és alt, però a mesura que s’executa aquest baixa. Això ens indica la carrega de treball desproporcionada que tenen els diferents threads. Alguns threads acaben el seu treball abans que altres per aquest motiu.

****

En aquest segon gràfic també es pot veure aquest detall per cada thread. el thread 9 té una càrrega major que els altres. Es pot veure que al final de l’execució molts threads estan inactius degut a aquesta càrrega de treball. Això ens mostra que no estem utilitzant bé el paral·lelisme i no estem aprofitant bé tots el threads.

**Strong Scalability**

****

Aquest gràfic ens mostra l’evolució del speedup en funció del nombre de threads utilitzats. S'observa que el speedup no creix linealment com ens agradaria. Això indica que el rendiment global no escala de manera proporcional al nombre de threads. Això és degut que alguns threads queden inactius mentre altres encara treballen, fet que s'ha observat també en els gràfics de Paraver. Això és causat per la distribució estàtica de les columnes, on algunes són més complexes que altres, creant un desequilibri de càrrega.

1D Block-Cyclic Geometric Data Decomposition by columns

Aquesta tècnica divideix les dades en blocs de grandària fixa i els distribueix de manera cíclica entre els fils disponibles. Cada fil processa columnes consecutives agrupades en blocs, amb un desplaçament cíclic que depèn del nombre total de fils i la grandària del bloc.

Calculem els blocs on cada fil comença amb un índex inicial basat en my\_\*thread \* BS i processa blocs consecutius, saltant num\_\*threads \* BS columnes en cada iteració.

A més ens assegurem que l'índex final px\_\*end no excedeixi el nombre total de columnes.

void mandel\_simple(int M[ROWS][COLS], double CminR, double CminI, double CmaxR, double CmaxI, double scale\_real, double scale\_imag, int maxiter)

{

// Calcular

int my\_thread= omp\_get\_thread\_num();

int num\_threads = omp\_get\_num\_threads();

for (int py = 0; py < ROWS; py++) {

for (int jj = my\_thread\*BS; jj<COLS; jj+=num\_threads\*BS) {

int px\_end = jj + BS;

if(px\_end > COLS) px\_end = COLS;

for(int px = jj; px<px\_end; px++) {

M[py][px] = pixel\_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, px, py, scale\_real, scale\_imag, maxiter);

if (output2histogram)

#pragma omp atomic

histogram[M[py][px] - 1]++;

if (output2display) {

/\* Scale color and display point \*/

#pragma omp critical

{

long color = (long)((M[py][px] - 1) \* scale\_color) + min\_color;

if (setup\_return == EXIT\_SUCCESS) {

XSetForeground(display, gc, color);

XDrawPoint(display, win, gc, px, py);

}

}

}

}

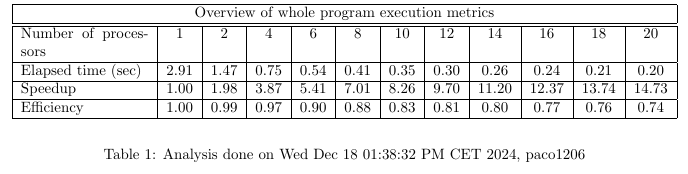
}

}

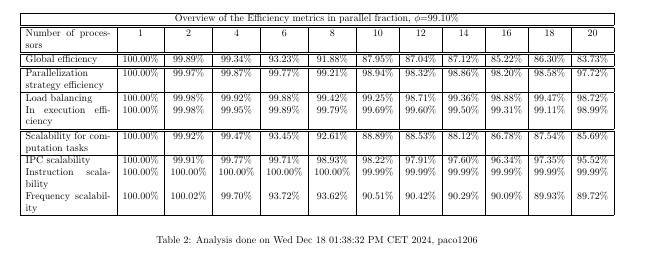
}



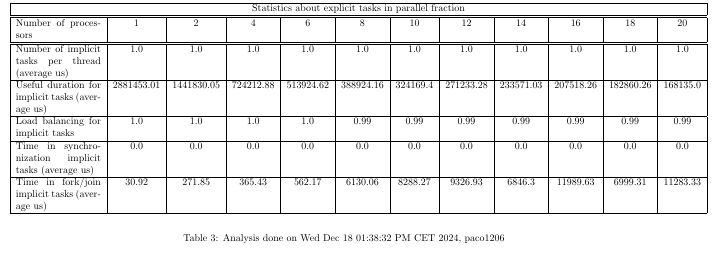
**Modelfactor analysis:**



En aquesta primera taula es pot observar que el temps d'execució disminueix notablement en comparació amb l'estratègia anterior, a més que el speedup ha augmentat força, fins a aproparnos a 15 amb 20 threads. Això indica que la descomposició en blocs cíclics ha millorat la distribució del treball i aprofitem millor la paral·lelització. A més la eficiencia l'hem millorat considerablement fent que només es redueixi un 16% de 1 a 20 threads

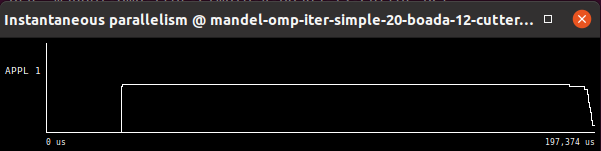


En aquesta segona taula podem confirmar el que veiem a la primera taula, la càrrega de treball ha millorat molt si ho comparem amb el primer codi. A més l'estratègia de paral·lelització també ha millorat considerablement.



En aquesta ultima taula podem veure el treball que ha fet cada thread i es pot observar que son xifres molt semblant, encara que podrien ser més iguals, indicant que aprofitem molt més el número de threads i que no hi hagin threads treballant mentre que d’altres sense fer res.

**Paraver Analysis:**

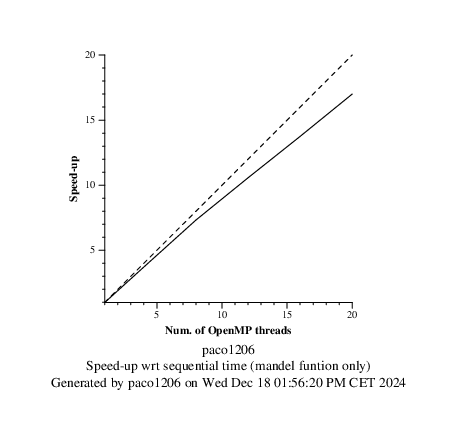
****

En aquest primer gràfic apreciem com tots els threads fan una quantitat de treball similar, veiem com inician tots al mateix temps i es mantenen treballant durant un temps, es veu en el gràfic la línia perfectament recta a una altura (quantitat de threads) i quan els threads van acaban les tasques, és quan veiem que la línia baixa cap al 0 de l’inici, com no totes les tasques acaban justament en el mateix moment no es una linea vertical de baixada pero s’apropa a ser-ho

****

En aquest segon gràfic de paraver, podem veure la duració de les tasques per threads. Es pot observar que gairebé tots els threads acaben quasi igual, i que no hi ha un desbalanceig de treball. El thread que més ràpid acaba és el primer.

**Strong Scalability**

****

En aquest últim gràfic podem veure com ara el speedup si creix correctament en funció als threads utilitzats. Encara que podria ser millor ja que a partir de aproximadament 8 threads no creix tant exponencialment.

1D Cyclic Geometric Data Decomposition by rows

En aquesta part busquem una tècnica semblant a l'anterior, però per files, les quals en la matriu es distribueixen cíclicament entre els threads disponibles. Cada thread processa un subconjunt de files no consecutives, amb un patró de salt que depèn del nombre de fils.

void mandel\_simple(int M[ROWS][COLS], double CminR, double CminI, double CmaxR, double CmaxI, double scale\_real, double scale\_imag, int maxiter)

{

int myid = omp\_get\_thread\_num();

int howmany = omp\_get\_num\_threads();

for (int i = myid; i < ROWS; i += howmany) {

for (int j = 0; j < COLS; j++) {

M[i][j] = pixel\_dwell(COLS, ROWS, CminR, CminI, CmaxR, CmaxI, j, i, scale\_real, scale\_imag, maxiter);

if (output2histogram)

#pragma omp atomic

histogram[M[i][j] - 1]++;

if (output2display) {

#pragma omp critical

{

long color = (long)((M[i][j] - 1) \* scale\_color) + min\_color;

if (setup\_return == EXIT\_SUCCESS) {

XSetForeground(display, gc, color);

XDrawPoint(display, win, gc, j, i);

}

}

}

}

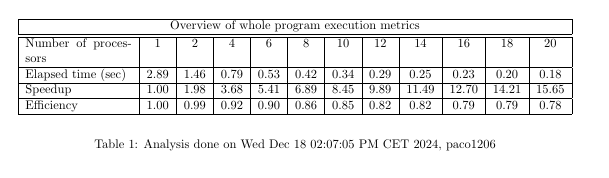
}

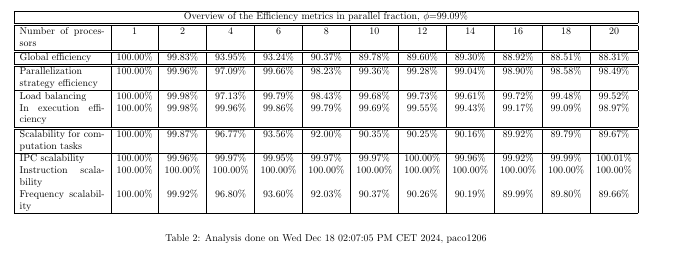
}

}

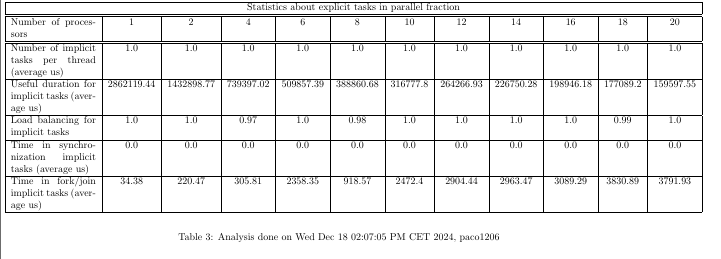


**Modelfactor analysis:**

En aquesta taula podem apreciar que el temps disminueix dràsticament en augmentar els processadors, aconseguint una millora significativa fins a 8 processadors. A partir de 10, el temps millora menys i tendeix a estabilitzar-se. Com ho fa també el speed up que sense cap límit, fins als 20 threads, augmenta significativament. L'eficiència cau progressivament conforme s'afegeixen processadors, des d'un 100% amb 1 fil fins a un 78% amb 20 processadors, lo qual podem considerar una bona eficiencia, encara així tenim marge de millora en el codi.



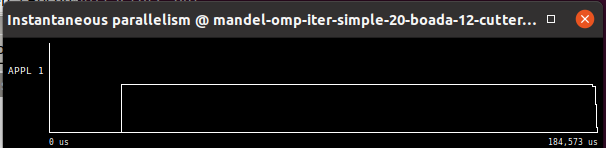
En aquesta segona taula veiem la bona eficiencia global sent d’un 88%, que es produida per la bona estrategia de paralelitzacio de més d’un 98% i el balanceig proper al 99%, per tant veiem que hem fet un bon codi en aquests nivells.



El nombre de tasques explícites per fil és constant (1 tasca per fil), la qual cosa suggereix que cada fil processa una única tasca.

El temps mitjà per tasca disminueix al principi, però eventualment s'estabilitza conforme el nombre de fils incrementa, pero sent molt de temps per tasca. Això ho podem relacionar amb la falta de fine-grain parallelism, que permetria dividir el treball en tasques més petites i balancejades.

**Paraver Analysis:**

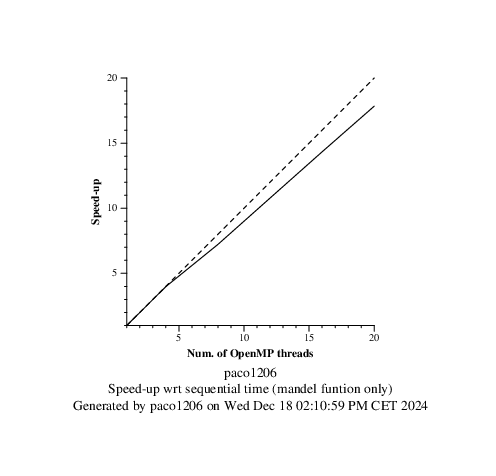
****

En aquest primer gràfic apreciem com tots els threads fan una quantitat de treball similar, veiem com inicien tots al mateix temps i es mantenen treballant durant un temps, es veu en el gràfic la línia perfectament recta a una altura (quantitat de threads) i quan els threads van acabant les tasques, és quan veiem que la línia baixa cap al 0 de l'inici, com no totes les tasques acaben justament en el mateix moment no és una línia vertical de baixada, però s'apropa a ser-ho. A diferència de l'anterior veiem que encara aquesta baixada és més vertical, el que vol dir que les tasques acaben en un temps més similar.

****

En aquest segon gràfic de paraver, podem veure la duració de les tasques per threads. Cada línia horitzontal representa un fil, i les barres vermelles indiquen el temps durant el qual cada fil executa una tasca. Es pot observar que tots els threads acaben quasi igual, i que no hi ha un desbalanceig de treball.

**Strong Scalability**



En aquest últim gràfic podem veure com creix el speed-up en funció del número de threads Encara que podria ser millor, ja que a partir d'aproximadament 5 threads cau una mica de la projecció ideal i, per tant, hi ha possibilitat de millora.