# HW2

> 112062706 林泳成


- HW2
  - Implementation
    - pthread
    - hybrid
  - Experiment & Analysis
    - Performance matrix & Speedup
    - Profile
    - Discussion
  - Conclusion


## Implementation

### pthread

```
int curX = 0, curY = 0, blockPerThread = 2500;

int jobRemain(){
    // Return available block size
    // Guarantee cur not pass limit
    int totalBlock = width * height;
    int curSpanX = curX + curY * width;
    if(curSpanX + blockPerThread < totalBlock)
        return blockPerThread;
    else if(curSpanX >= totalBlock)
        return -1;
    else
        return totalBlock - curSpanX;
}
```

```
void* calculate(void*){
    while(true){
        pthread_mutex_lock(&mutex);
        int block = jobRemain();
        int startX, startY;
        if(block == -1){
            pthread_mutex_unlock(&mutex);
            break;
        }
        else{
            startX = curX;
            startY = curY;
            curX += block;
            while(curX >= width){
                curX -= width;
                ++curY;
            }
            pthread_mutex_unlock(&mutex);
        }
```

每個thread都會在while迴圈內去拿新的工作，工作大小定義在global variable(blockPerThread)。每個thread都會一直拿工作直到沒有新的工作可以拿。jobRemain function會回傳可以拿的資料量。另外有用curX和curY指向下一個工作的起始位置，所以會用lock把這個curX和curY的更新包起來。

```
if(block != blockPerThread){
    int iter = 0;
    while(true){
        if(iter == block)
            break;
        int repeats = 0;
        double x = 0;
        double y = 0;
        double length_squared = 0;
        double y0 = startY * ((upper - lower) / height) + lower;
        double x0 = startX * ((right - left) / width) + left;
        while (repeats < iters && length_squared < 4) {
            double temp = x * x - y * y + x0;
            y = 2 * x * y + y0;
            x = temp;
            length_squared = x * x + y * y;
            ++repeats;
        }
        image[startY * width + startX] = repeats;
        ++startX;
        if(startX == width){
            ++startY;
            startX = 0;
        }
        ++iter;
    }
    break;
}
```

如果拿到的工作是最後一份，那就讓他直接執行

剩下的部分會用vectorization實作

```
double startXVec[8], startYVec[8], repeatCpy[8], startYCpy[8], startXCpy[8], xCpy[8], yCpy[8], lengthSquaredCpy[8], x0Cpy[8], y0Cpy[8];
for(int i=0;i<8;++i){
    // ++iter;
    startXVec[i] = (double)startX;
    startYVec[i] = (double)startY;
    ++startX;
    if(startX >= width){
        startX = 0;
        ++startY;
    }
}
__m512d mxVec = _mm512_set1_pd(0.0);
__m512d myVec = _mm512_set1_pd(0.0);
__m512d mstartXVec = _mm512_loadu_pd(startXVec);
__m512d mstartYVec = _mm512_loadu_pd(startYVec);
__m512d mlengthSquared = _mm512_set1_pd(0.0);
__m512d mupperVec = _mm512_set1_pd(upper);
__m512d mlowerVec = _mm512_set1_pd(lower);
__m512d mrightVec = _mm512_set1_pd(right);
__m512d mleftVec = _mm512_set1_pd(left);
__m512d mheightVec = _mm512_set1_pd((double)height);
__m512d mwidthVec = _mm512_set1_pd((double)width);

__m512d mitersVec = _mm512_set1_pd((double)iters);
__m512d mrepeatsVec = _mm512_set1_pd(0.0);

__m512d mx0Vec = _mm512_fmadd_pd(mstartXVec, _mm512_div_pd(_mm512_sub_pd(mrightVec, mleftVec), mwidthVec), mleftVec);
__m512d my0Vec = _mm512_fmadd_pd(mstartYVec, _mm512_div_pd(_mm512_sub_pd(mupperVec, mlowerVec), mheightVec), mlowerVec);
```

先把前8個資料塞進__m512d的vector，之後就同時更新8個變數

```c
while(iter < block){
    __mmask8 mask4 = _mm512_cmp_pd_mask(mlengthSquared, _mm512_set1_pd(4.0), _CMP_GE_OQ);
    __mmask8 maskRepeats = _mm512_cmp_pd_mask(mrepeatsVec, mitersVec, _CMP_GE_OQ);
    __mmask8 maskEmpty = _mm512_cmp_pd_mask(mrepeatsVec, _mm512_set1_pd(-1.0), _CMP_EQ_OQ);
    for(idx=0;idx<8;++idx){
        if(((mask4 & (1 << idx)) || (maskRepeats & (1 << idx))) && !(maskEmpty & (1 << idx))){
            _mm512_store_pd(&xCpy, mxVec);
            _mm512_store_pd(&yCpy, myVec);
            _mm512_store_pd(&x0Cpy, mx0Vec);
            _mm512_store_pd(&y0Cpy, my0Vec);
            _mm512_store_pd(&repeatCpy, mrepeatsVec);
            _mm512_store_pd(&startXCpy, mstartXVec);
            _mm512_store_pd(&startYCpy, mstartYVec);
            image[(int)startYCpy[idx] * width + (int)startXCpy[idx]] = (int)repeatCpy[idx];
            if(iter == block)
                break;
            else{
                ++iter;
                _mm512_store_pd(&xCpy, mxVec);
                _mm512_store_pd(&yCpy, myVec);
                _mm512_store_pd(&x0Cpy, mx0Vec);
                _mm512_store_pd(&y0Cpy, my0Vec);
                _mm512_store_pd(&lengthSquaredCpy, mlengthSquared);
                repeatCpy[idx] = 0.0;
                startXCpy[idx] = (double)startX;
                startYCpy[idx] = (double)startY;
                xCpy[idx] = 0.0;
                yCpy[idx] = 0.0;
                lengthSquaredCpy[idx] = 0.0;
                x0Cpy[idx] = startX * ((right - left) / width) + left;
                y0Cpy[idx] = startY * ((upper - lower) / height) + lower;
                mrepeatsVec = _mm512_load_pd(&repeatCpy);
                mstartXVec = _mm512_load_pd(&startXCpy);
                mstartYVec = _mm512_load_pd(&startYCpy);
                mxVec = _mm512_load_pd(&xCpy);
                myVec = _mm512_load_pd(&yCpy);
                mlengthSquared = _mm512_load_pd(&lengthSquaredCpy);
                mx0Vec = _mm512_load_pd(&x0Cpy);
                my0Vec = _mm512_load_pd(&y0Cpy);
                ++startX;
```

```c
                ++startX;
                if(startX >= width){
                    startX = 0;
                    ++startY;
                }
            }
        }
    }
    __m512d tmpVec = _mm512_add_pd(_mm512_sub_pd(_mm512_mul_pd(mxVec, mxVec), _mm512_mul_pd(myVec, myVec)), mx0Vec);
    myVec = _mm512_add_pd(_mm512_mul_pd(_mm512_set1_pd(2.0), _mm512_mul_pd(mxVec, myVec)), my0Vec);
    mxVec = tmpVec;
    mlengthSquared = _mm512_add_pd(_mm512_mul_pd(mxVec, mxVec), _mm512_mul_pd(myVec, myVec));
    mrepeatsVec = _mm512_add_pd(mrepeatsVec, _mm512_set1_pd(1.0));
}
```

每次都去檢查看看有沒有任務跑完了，如果有就把那一格的repeats次數寫出來，並且把新的工作覆蓋在空出來的格子上

```
for(int i=0;i<8;++i){
    if(i == idx)
        continue;
    int repeats = 0;
    double x = 0;
    double y = 0;
    double length_squared = 0;
    double y0 = (int)startYCpy[i] * ((upper - lower) / height) + lower;
    double x0 = (int)startXCpy[i] * ((right - left) / width) + left;
    while (repeats < iters && length_squared < 4) {
        double temp = x * x - y * y + x0;
        y = 2 * x * y + y0;
        x = temp;
        length_squared = x * x + y * y;
        ++repeats;
    }
    image[(int)startYCpy[i] * width + (int)startXCpy[i]] = repeats;
```

直到剩下最後7個任務無法繼續用vectorization，就重新sequential得跑

**優化後**

```
__mmask8 maskEmpty = _mm512_cmp_pd_mask(mrepeatsVec, _mm512_set1_pd(-1.0), _CMP_EQ_OQ);
for(idx=0;idx<8;++idx){
    if((mask4 & (1 << idx)) || (maskRepeats & (1 << idx))) && !(maskEmpty & (1 << idx))){
```

其實後來發現vectorization不一定要全滿才能做，裡面也可以塞無用的空直，這樣就不需要重跑最後7個任務了(這個優化因為是每次拿新工作就會在最後執行到，所以跑judge的時候總共快了16秒左右)

```
for(int i=0;i<min(8, block);++i){
    // ++iter;
    startXVec[i] = (double)startX;
    startYVec[i] = (double)startY;
    ++startX;
    if(startX >= width){
        startX = 0;
        ++startY;
    }
}
```

```
if(block != blockPerThread)
    break;
```

還有一個小優化是在最後拿不滿blockPerThread的工作量時才會做的sequencial run。如果也把這些塞不滿8格的工作也拿去做vectorization就可以不需要一個一個慢慢做(這個優化只優化最後一個工作，所以進步幅度有限，最後跑judge總共也只快了0.3秒)

```
// __mmask8 maskEmpty = _mm512_cmp_pd_mask(mrepeatsVec, _mm512_set1_pd(-1.0), _CMP_EQ_OQ);
for(idx=0;idx<8;++idx){
    if((mask4 & (1 << idx)) || (maskRepeats & (1 << idx))){
```

後來發現如果圖片足夠大，那最後7個點的repeats次數一定會比重新計算(curX=0, curY+=1)還要早

算完，因此就不需要另外把不合法的工作排除，也就不需要開mask來遮掉有問題的工作(跑judge快了2秒)

## hybrid

```
int floorHeight = height / size;
int extraRows = height % size;
int rows = rank < extraRows ? floorHeight + 1 : floorHeight;
```

每個process都會分到rows行的工作。如果無法整除，前面幾個process會拿到多一行的工作

```
#pragma omp parallel num_threads(CPU_COUNT(&cpu_set))
{
    int startX = 0, startY = 0;
    while(startY < rows){
        int idx = -1;
        int iter = 0;

        double startXVec[8], startYVec[8], repeatCpy[8], startYCpy[8], startXCpy[8], xCpy[8], yCpy[8], lengthSquaredCpy[8], x0Cpy[8], y0Cpy[8];
        for(int i=0;i<8;++i){
            ++iter;
            startXVec[i] = (double)startX;
            startYVec[i] = (double)startY;
            ++startX;
        }
        __m512d mxVec = _mm512_set1_pd(0.0);
        __m512d myVec = _mm512_set1_pd(0.0);
        __m512d mstartXVec = _mm512_loadu_pd(startXVec);
        __m512d mstartYVec = _mm512_loadu_pd(startYVec);
        __m512d mlengthSquared = _mm512_set1_pd(0.0);
        __m512d mupperVec = _mm512_set1_pd(upper);
        __m512d mlowerVec = _mm512_set1_pd(lower);
        __m512d mrightVec = _mm512_set1_pd(right);
        __m512d mleftVec = _mm512_set1_pd(left);
        __m512d mheightVec = _mm512_set1_pd((double)height);
        __m512d mwidthVec = _mm512_set1_pd((double)width);

        __m512d mitersVec = _mm512_set1_pd((double)iters);
        __m512d mrepeatsVec = _mm512_set1_pd(0.0);

        __m512d mx0Vec = _mm512_fmadd_pd(mstartXVec, _mm512_div_pd(_mm512_sub_pd(mrightVec, mleftVec), mwidthVec), mleftVec);
        __m512d my0Vec = _mm512_fmadd_pd(_mm512_add_pd(_mm512_mul_pd(mstartYVec, _mm512_set1_pd((double)size)), _mm512_set1_pd((double)rank)), _mm512_div_pd(_

        while(iter < width){
            __mmask8 mask4 = _mm512_cmp_pd_mask(mlengthSquared, _mm512_set1_pd(4.0), _CMP_GE_OQ);
            __mmask8 maskRepeats = _mm512_cmp_pd_mask(mrepeatsVec, mitersVec, _CMP_GE_OQ);
            for(idx=0;idx<8;++idx){
                if((mask4 & (1 << idx)) || (maskRepeats & (1 << idx))){
                    _mm512_store_pd(&xCpy, mxVec);
                    _mm512_store_pd(&yCpy, myVec);
                    _mm512_store_pd(&x0Cpy, mx0Vec);
                    _mm512_store_pd(&y0Cpy, my0Vec);
                    _mm512_store_pd(&repeatCpy, mrepeatsVec);
```

OpenMP開出來的thread執行的工作基本和pthread一樣，都用vectorization去優化計算

每個process實際上會是輪流拿每一行的工作

```
            _mm512_store_pd(&repeatCpy, mrepeatsVec);
            _mm512_store_pd(&startXCpy, mstartXVec);
            _mm512_store_pd(&startYCpy, mstartYVec);
            image[(int)startYCpy[idx] * width + (int)startXCpy[idx]] = (int)repeatCpy[idx];
            if(iter == width)
                break;
            else{
                ++iter;
                _mm512_store_pd(&xCpy, mxVec);
                _mm512_store_pd(&yCpy, myVec);
                _mm512_store_pd(&x0Cpy, mx0Vec);
                _mm512_store_pd(&y0Cpy, my0Vec);
                _mm512_store_pd(&lengthSquaredCpy, mlengthSquared);
                repeatCpy[idx] = 0.0;
                startXCpy[idx] = (double)startX;
                startYCpy[idx] = (double)startY;
                xCpy[idx] = 0.0;
                yCpy[idx] = 0.0;
                lengthSquaredCpy[idx] = 0.0;
                x0Cpy[idx] = startX * ((right - left) / width) + left;
                y0Cpy[idx] = (((double)startY*size) + rank) * ((upper - lower) / height) + lower;
                mrepeatsVec = _mm512_load_pd(&repeatCpy);
                mstartXVec = _mm512_load_pd(&startXCpy);
                mstartYVec = _mm512_load_pd(&startYCpy);
                mxVec = _mm512_load_pd(&xCpy);
                myVec = _mm512_load_pd(&yCpy);
                mlengthSquared = _mm512_load_pd(&lengthSquaredCpy);
                mx0Vec = _mm512_load_pd(&x0Cpy);
                my0Vec = _mm512_load_pd(&y0Cpy);
                ++startX;
            }
        }
    }
    __m512d tmpVec = _mm512_add_pd(_mm512_sub_pd(_mm512_mul_pd(mxVec, mxVec), _mm512_mul_pd(myVec, myVec)), mx0Vec);
    myVec = _mm512_add_pd(_mm512_mul_pd(_mm512_set1_pd(2.0), _mm512_mul_pd(mxVec, myVec)), my0Vec);
    mxVec = tmpVec;
    mlengthSquared = _mm512_add_pd(_mm512_mul_pd(mxVec, mxVec), _mm512_mul_pd(myVec, myVec));
    mrepeatsVec = _mm512_add_pd(mrepeatsVec, _mm512_set1_pd(1.0));
}
```

```
for(int i=0;i<8;++i){
    if(i == idx)
        continue;
    int repeats = 0;
    double x = 0;
    double y = 0;
    double length_squared = 0;
    double y0 = ((int)startYCpy[i]*size+rank) * ((upper - lower) / height) + lower;
    double x0 = (int)startXCpy[i] * ((right - left) / width) + left;
    while (repeats < iters && length_squared < 4) {
        double temp = x * x - y * y + x0;
        y = 2 * x * y + y0;
        x = temp;
        length_squared = x * x + y * y;
        ++repeats;
    }
    image[(int)startYCpy[i] * width + (int)startXCpy[i]] = repeats;
}
++startY;
startX = 0;
```

剩餘無法塞進__m512d的工作就在最後用sequencial去做

```
displs[0] = 0;
for(int i = 0;i < size;i++){
    if(i < extraRows)
        recvcounts[i] = (floorHeight + 1) * width;
    else
        recvcounts[i] = floorHeight * width;

    if(i != 0)
        displs[i] = displs[i-1] + recvcounts[i-1];
}
```

```
MPI_Gatherv(image, rows*width, MPI_INT, gatherImage, recvcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);
```

由於每個process是跳著拿工作的，且工作量都不一樣，所以直接用MPI_Gather不是個好方法
因此使用MPI_Gatherv來整合整張圖
MPI_Gatherv會需要多吃兩個參數，一個是用來存每個rank要傳的data量，一個是buffer接收每個rank的資料時的起始位置

```
if(rank==0){
    int* final_image = (int*)malloc(width * height * sizeof(int));
    clock_gettime(CLOCK_MONOTONIC, &start_t);
    int index = 0;
    for(int i = 0;i < rows;i++){
        int stride = rows;
        int extraRowLeft = extraRows;
        for(int row = i; row < height && index < height*width; row += stride){
            for(int col = 0; col < width; col++){
                final_image[index] = gatherImage[row*width + col];
                index++;
            }
            --extraRowLeft;
            if(extraRowLeft < 0){
                stride = floorHeight;
            }
        }
    }
```

Gather完的圖片會是每個process跳著拿工作的結果
所以需要再跳著把圖片組回來

**優化後**

```
#pragma omp parallel num_threads(CPU_COUNT(&cpu_set))
{
    double startXVec[8], startYVec[8], repeatCpy[8], startYCpy[8], startXCpy[8], xCpy[8], yCpy[8], lengthSquaredCpy[8], x0Cpy[8], y0Cpy[8];
    #pragma omp for schedule(static, 10)
    for(int startY = 0;startY < rows;++startY){
        int startX = 0;
        for(int i=0;i<8;++i){
            startXVec[i] = (double)startX;
            startYVec[i] = (double)startY;
            ++startX;
        }
        __m512d mxVec = _mm512_set1_pd(0.0);
        __m512d myVec = _mm512_set1_pd(0.0);
        __m512d mstartXVec = _mm512_loadu_pd(startXVec);
        __m512d mstartYVec = _mm512_loadu_pd(startYVec);
        __m512d mlengthSquared = _mm512_set1_pd(0.0);
        __m512d mupperVec = _mm512_set1_pd(upper);
        __m512d mlowerVec = _mm512_set1_pd(lower);
        __m512d mrightVec = _mm512_set1_pd(right);
        __m512d mleftVec = _mm512_set1_pd(left);
        __m512d mheightVec = _mm512_set1_pd((double)height);
        __m512d mwidthVec = _mm512_set1_pd((double)width);

        __m512d mitersVec = _mm512_set1_pd((double)iters);
        __m512d mrepeatsVec = _mm512_set1_pd(0.0);

        __m512d mx0Vec = _mm512_fmadd_pd(mstartXVec, _mm512_div_pd(_mm512_sub_pd(mrightVec, mleftVec), mwidthVec), mleftVec);
        __m512d my0Vec = _mm512_fmadd_pd(_mm512_add_pd(_mm512_mul_pd(mstartYVec, _mm512_set1_pd((double)size)), _mm512_set1_pd((double)rank)), _mm512_div_p
        int idx = -1;
        for(int iter = 8;iter < width;++iter){
            while(true){
                __mmask8 mask4 = _mm512_cmp_pd_mask(mlengthSquared, _mm512_set1_pd(4.0), _CMP_GE_OQ);
                __mmask8 maskRepeats = _mm512_cmp_pd_mask(mrepeatsVec, mitersVec, _CMP_GE_OQ);
                bool breakFlag = false;
                for(idx=0;idx<8;++idx){
                    if((mask4 & (1 << idx)) || (maskRepeats & (1 << idx))){
                        _mm512_store_pd(&xCpy, mxVec);
                        _mm512_store_pd(&yCpy, myVec);
                        _mm512_store_pd(&x0Cpy, mx0Vec);
                        _mm512_store_pd(&y0Cpy, my0Vec);
                        _mm512_store_pd(&repeatCpy, mrepeatsVec);
                        _mm512_store_pd(&startXCpy, mstartXVec);
                        _mm512_store_pd(&startYCpy, mstartYVec);
```

因為OpenMP的data parallelism需要把程式改成互相independent的for迴圈，所以我把原本的程式改成能用for迴圈來增加startY，讓原本會有dependecy的程式能分離，讓OpenMP自動做工作分配(我原本跑judge要303秒，這個優化讓這隻程式變成跑judge只需要127秒。**突然發現OpenMP很吃programmer的使用**)

# Experiment & Analysis

## Performance matrix & Speedup

### pthread

因為看到strict28到strict36都是一樣的iteration、圖寬和圖高，就只差在x0, x1, y0, y1，就想說拿來當作實驗input。結果發現時間差很多，最快的6秒就完成了，最慢的要107秒。

| Input file | strict28.txt | strict29.txt | strict30.txt | strict31.txt | strict32.txt | strict33.txt | strict34.txt | strict35.txt | strict36.txt |
|---|---|---|---|---|---|---|---|---|---|
| x0 | -0.6743255349 | 0.2936331265 | -0.3421054598 | -0.3070888275 | -0.5506211524 | -0.2989925066 | -0.5506164692 | -0.2931209325 | -0.2872724083 |
| x1 | -0.674237311 | 0.2938268309 | -0.2373971479 | -0.247606819 | -0.5506132349 | -0.2772002993 | -0.5506164628 | -0.274142734 | -0.2791226113 |
| y0 | 0.3623527742 | -0.01494847713 | -0.6373595233 | -0.6245844142 | 0.6273469513 | -0.6327591639 | 0.6273445437 | -0.6337125743 | -0.6345413373 |
| y1 | 0.362305991 | -0.01505119437 | -0.6884105744 | -0.6535851592 | 0.6273427528 | -0.6433840615 | 0.6273445404 | -0.6429654881 | -0.6385148108 |
| Process | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Thread | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Initialization time | 0.000019 | 0.00002 | 0.000018 | 0.00002 | 0.000022 | 0.00002 | 0.000019 | 0.000017 | 0.00002 |
| Thread compute time | 7.023348 | 24.733721 | 6.139337 | 69.2553 | 20.96561 | 83.905639 | 106.299448 | 46.753523 | 20.139368 |
| Write PNG time | 1.155882 | 0.947597 | 0.762026 | 0.968045 | 1.48397 | 0.900907 | 1.541206 | 1.000165 | 0.95577 |
| Total time | 8.179249 | 25.681338 | 6.901381 | 70.223365 | 22.449602 | 84.806566 | 107.840673 | 47.753705 | 21.095158 |
| Process | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Thread | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Initialization time | 0.000018 | 0.00002 | 0.000021 | 0.000022 | 0.000019 | 0.000018 | 0.000018 | 0.000024 | 0.000033 |
| Thread compute time | 3.523751 | 12.364939 | 3.080458 | 34.568468 | 10.484671 | 41.942321 | 53.135242 | 23.343529 | 10.070225 |
| Write PNG time | 1.152939 | 0.946612 | 0.765715 | 0.961856 | 1.481843 | 0.901876 | 1.539031 | 0.995231 | 0.96711 |
| Total time | 4.676708 | 13.311571 | 3.846194 | 35.530346 | 11.966533 | 42.844215 | 54.674291 | 24.338784 | 11.037368 |
| Process | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Thread | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Initialization time | 0.000019 | 0.000021 | 0.000019 | 0.00002 | 0.000022 | 0.000021 | 0.000018 | 0.000022 | 0.000019 |
| Thread compute time | 2.351429 | 8.252186 | 2.05586 | 23.07916 | 6.987645 | 27.976962 | 35.483878 | 15.578621 | 6.716372 |
| Write PNG time | 1.162455 | 0.947836 | 0.760895 | 0.968042 | 1.48168 | 0.900743 | 1.540511 | 1.001089 | 0.963097 |
| Total time | 3.513903 | 9.200043 | 2.816774 | 24.047222 | 8.469347 | 28.877726 | 37.024407 | 16.579732 | 7.679488 |
| Process | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Thread | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Initialization time | 0.000035 | 0.000021 | 0.000019 | 0.000022 | 0.000018 | 0.00002 | 0.000019 | 0.000022 | 0.000041 |
| Thread compute time | 1.766399 | 6.187515 | 1.551395 | 17.285039 | 5.232121 | 21.007626 | 26.625018 | 11.698381 | 5.036169 |
| Write PNG time | 1.157113 | 0.94754 | 0.753115 | 0.960356 | 1.483474 | 0.897758 | 1.532625 | 1.001179 | 0.958111 |
| Total time | 2.923547 | 7.135076 | 2.304529 | 18.245417 | 6.715613 | 21.905404 | 28.157662 | 12.699582 | 5.994321 |
| Process | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Thread | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Initialization time | 0.000035 | 0.000033 | 0.000041 | 0.00002 | 0.000031 | 0.000019 | 0.00002 | 0.000019 | 0.000021 |
| Thread compute time | 1.41486 | 4.95212 | 1.237492 | 13.837364 | 4.193321 | 16.804226 | 21.30085 | 9.357604 | 4.030731 |
| Write PNG time | 1.160397 | 0.942978 | 0.756135 | 0.956834 | 1.4834 | 0.898769 | 1.53933 | 1.002529 | 0.962724 |
| Total time | 2.575292 | 5.895131 | 1.993668 | 14.794218 | 5.676752 | 17.703014 | 22.8402 | 10.360152 | 4.993476 |
| Process | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Thread | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Initialization time | 0.00003 | 0.000034 | 0.000039 | 0.000025 | 0.000029 | 0.000029 | 0.00004 | 0.000027 | 0.000031 |
| Thread compute time | 7.028638 | 24.740941 | 6.137511 | 69.227421 | 20.953597 | 84.014062 | 106.520144 | 46.712085 | 20.129814 |
| Write PNG time | 1.16818 | 0.950249 | 0.762754 | 0.970395 | 1.48905 | 0.905065 | 1.546626 | 1.005642 | 0.968014 |
| Total time | 8.196848 | 25.691224 | 6.900304 | 70.197841 | 22.442676 | 84.919156 | 108.06681 | 47.717754 | 21.097859 |
| Process | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Thread | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Initialization time | 0.00005 | 0.000033 | 0.000033 | 0.00003 | 0.000027 | 0.000028 | 0.000036 | 0.000038 | 0.000028 |
| Thread compute time | 7.028272 | 24.736316 | 6.138182 | 69.184104 | 20.94536 | 84.023794 | 106.468744 | 46.785784 | 20.131487 |
| Write PNG time | 1.174938 | 0.954152 | 0.761314 | 0.967213 | 1.48566 | 0.903366 | 1.544881 | 1.014395 | 0.967731 |
| Total time | 8.20326 | 25.690501 | 6.899529 | 70.151347 | 22.431047 | 84.927188 | 108.013661 | 47.800217 | 21.099246 |
| Process | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| Thread | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Initialization time | 0.000029 | 0.000028 | 0.00005 | 0.000029 | 0.000028 | 0.000031 | 0.000021 | 0.000027 | 0.000396 |
| Thread compute time | 7.028294 | 24.745348 | 6.143538 | 69.226752 | 20.943351 | 84.004106 | 106.502764 | 46.77548 | 20.117233 |
| Write PNG time | 1.1645 | 0.953958 | 0.762294 | 0.977723 | 1.482349 | 0.906031 | 1.568923 | 1.0155 | 0.965468 |
| Total time | 8.192823 | 25.699334 | 6.905882 | 70.204504 | 22.425728 | 84.910168 | 108.071708 | 47.791007 | 21.083097 |
| Process | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Thread | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Initialization time | 0.000028 | 0.000029 | 0.000034 | 0.000028 | 0.000036 | 0.000041 | 0.000049 | 0.000031 | 0.000023 |
| Thread compute time | 7.031562 | 24.755604 | 6.143788 | 69.20522 | 20.949939 | 84.010889 | 106.506712 | 46.810372 | 20.141182 |
| Write PNG time | 1.188505 | 0.958572 | 0.762239 | 0.971541 | 1.520258 | 0.912169 | 1.577946 | 1.017053 | 0.965183 |
| Total time | 8.220095 | 25.714105 | 6.906061 | 70.176789 | 22.470233 | 84.923099 | 108.084707 | 47.827456 | 21.106388 |

實驗簡單測了各個input在不同數量thread和process的情況下能加速多少

實驗結果也明確顯示multi-process在這種實作下是無法加速的(因為只有使用pthread)

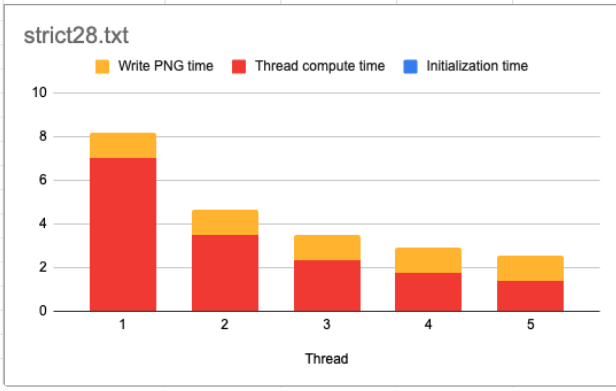
上圖中multi-process的各個執行時間是取所有process中最慢的

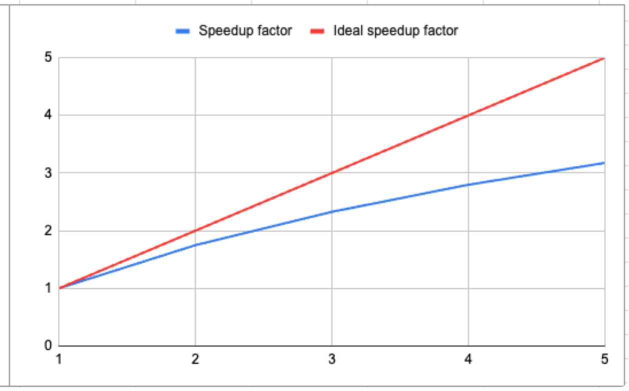
其中**Initialization time**用來計算初始化區域變數的時間

**Thread computation time**用來計算從fork所有thread到join完所有thread所花時間

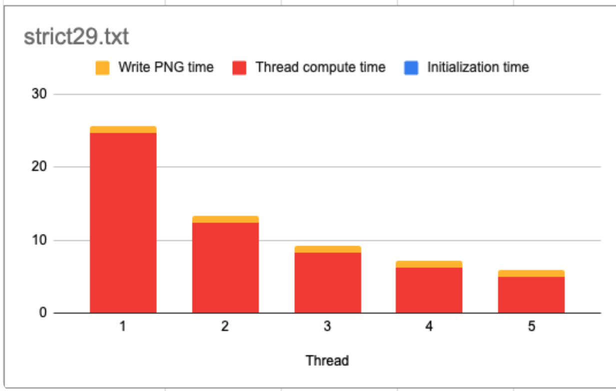
**Write PNG time**用來計算最後畫圖的時間

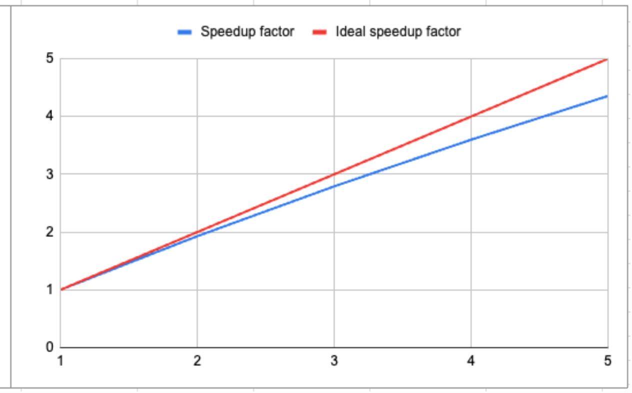
| Thread | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Initialization time | 0.000019 | 0.000018 | 0.000019 | 0.000035 | 0.000035 |
| Thread compute time | 7.023348 | 3.523751 | 2.351429 | 1.766399 | 1.41486 |
| Write PNG time | 1.155882 | 1.152939 | 1.162455 | 1.157113 | 1.160397 |
| Total time | 8.179249 | 4.676708 | 3.513903 | 2.923547 | 2.575292 |
| Speedup factor | 1 | 1.74893301 | 2.327682067 | 2.797714215 | 3.176047221 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |



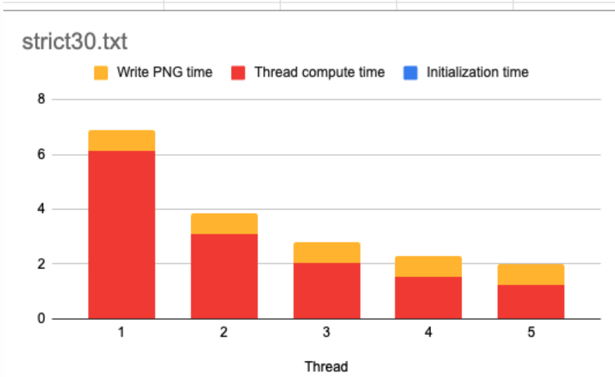



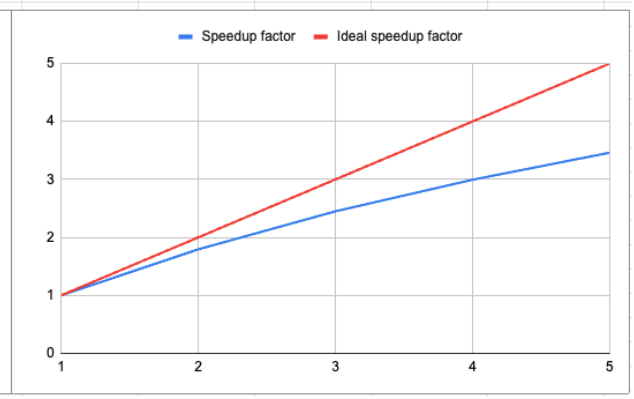

| Thread | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Initialization time | 0.00002 | 0.00002 | 0.000021 | 0.000021 | 0.000033 |
| Thread compute time | 24.733721 | 12.364939 | 8.252186 | 6.187515 | 4.95212 |
| Write PNG time | 0.947597 | 0.946612 | 0.947836 | 0.94754 | 0.942978 |
| Total time | 25.681338 | 13.311571 | 9.200043 | 7.135076 | 5.895131 |
| Speedup factor | 1 | 1.929249222 | 2.791436736 | 3.599308262 | 4.356364261 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |



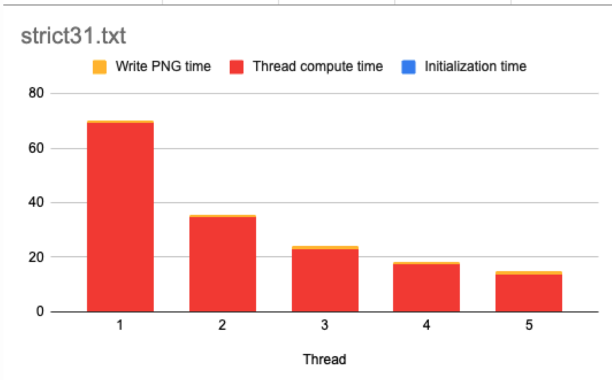



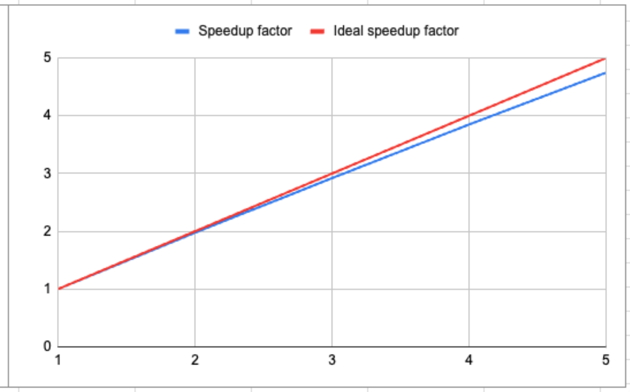

| Thread | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Initialization time | 0.000018 | 0.000021 | 0.000019 | 0.000019 | 0.000041 |
| Thread compute time | 6.139337 | 3.080458 | 2.05586 | 1.551395 | 1.237492 |
| Write PNG time | 0.762026 | 0.765715 | 0.760895 | 0.753115 | 0.756135 |
| Total time | 6.901381 | 3.846194 | 2.816774 | 2.304529 | 1.993668 |
| Speedup factor | 1 | 1.794340327 | 2.450101073 | 2.994703473 | 3.461650084 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |



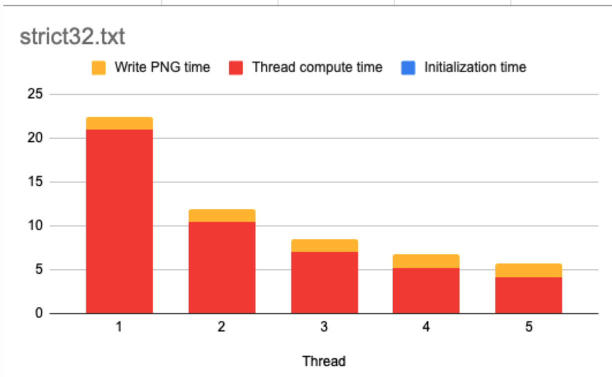



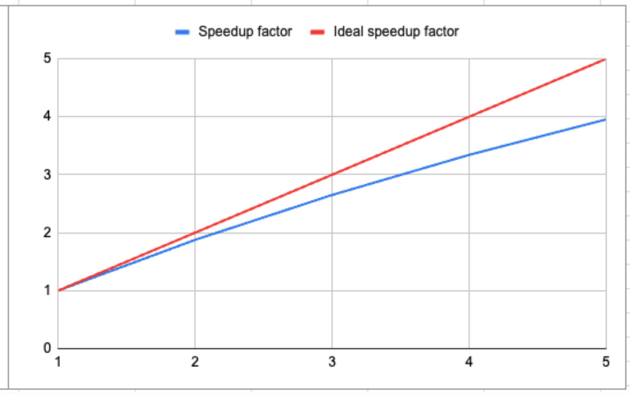
| | | | | | |
|---|---|---|---|---|---|
| Thread | 1 | 2 | 3 | 4 | 5 |
| Initialization time | 0.00002 | 0.000022 | 0.00002 | 0.000022 | 0.00002 |
| Thread compute time | 69.2553 | 34.568468 | 23.07916 | 17.285039 | 13.837364 |
| Write PNG time | 0.968045 | 0.961856 | 0.968042 | 0.960356 | 0.956834 |
| Total time | 70.223365 | 35.530346 | 24.047222 | 18.245417 | 14.794218 |
| Speedup factor | 1 | 1.976433469 | 2.92022775 | 3.848822145 | 4.74667637 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |



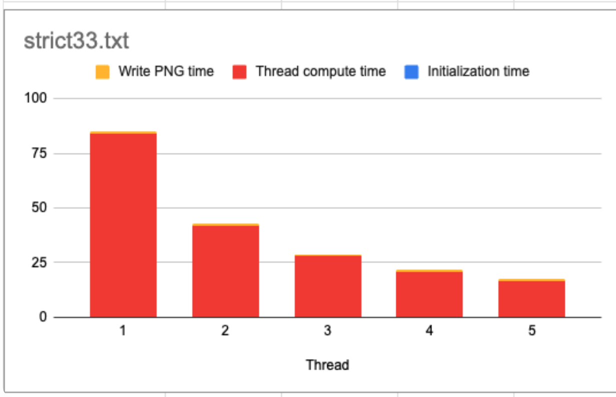



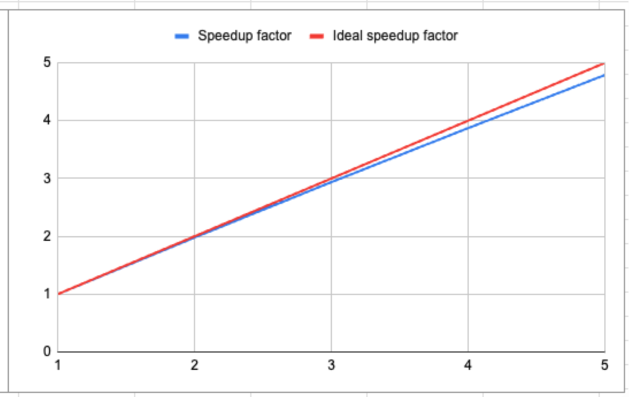

| | | | | | |
|---|---|---|---|---|---|
| Thread | 1 | 2 | 3 | 4 | 5 |
| Initialization time | 0.000022 | 0.000019 | 0.000022 | 0.000018 | 0.000031 |
| Thread compute time | 20.96561 | 10.484671 | 6.987645 | 5.232121 | 4.193321 |
| Write PNG time | 1.48397 | 1.481843 | 1.48168 | 1.483474 | 1.4834 |
| Total time | 22.449602 | 11.966533 | 8.469347 | 6.715613 | 5.676752 |
| Speedup factor | 1 | 1.876032264 | 2.650688654 | 3.342896918 | 3.954656113 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |



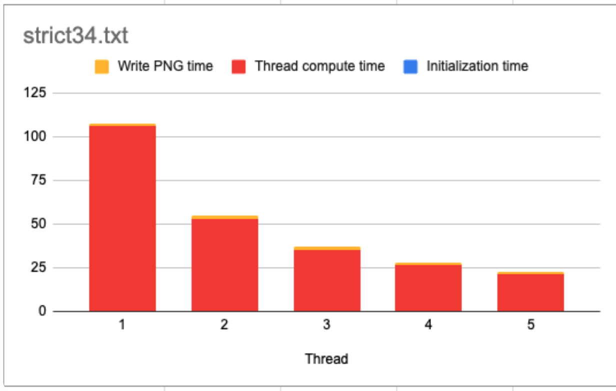



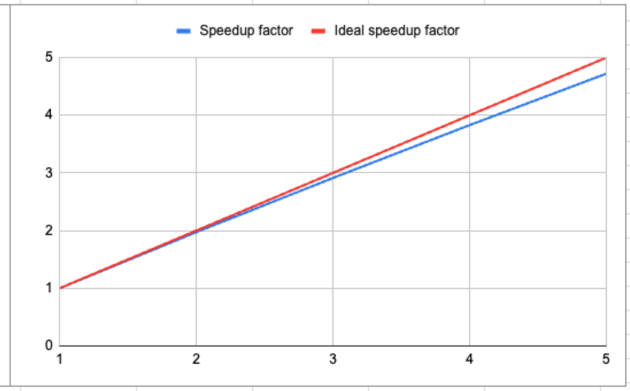

| | | | | | |
|---|---|---|---|---|---|
| Thread | 1 | 2 | 3 | 4 | 5 |
| Initialization time | 0.00002 | 0.000018 | 0.000021 | 0.00002 | 0.000019 |
| Thread compute time | 83.905639 | 41.942321 | 27.976962 | 21.007626 | 16.804226 |
| Write PNG time | 0.900907 | 0.901876 | 0.900743 | 0.897758 | 0.898769 |
| Total time | 84.806566 | 42.844215 | 28.877726 | 21.905404 | 17.703014 |
| Speedup factor | 1 | 1.979416964 | 2.936746682 | 3.871490615 | 4.790515672 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |



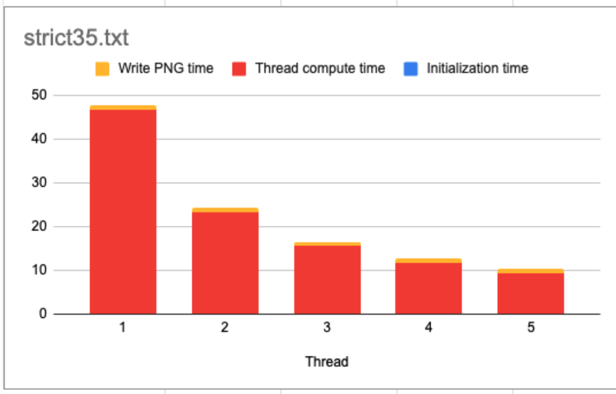



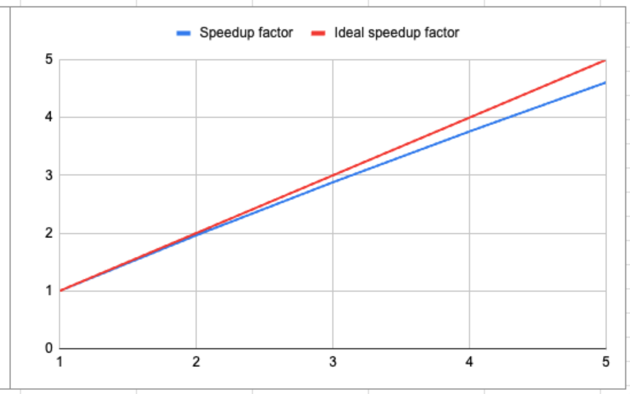
| Thread | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Initialization time | 0.000019 | 0.000018 | 0.000018 | 0.000019 | 0.00002 |
| Thread compute time | 106.299448 | 53.135242 | 35.483878 | 26.625018 | 21.30085 |
| Write PNG time | 1.541206 | 1.539031 | 1.540511 | 1.532625 | 1.53933 |
| Total time | 107.840673 | 54.674291 | 37.024407 | 28.157662 | 22.8402 |
| Speedup factor | 1 | 1.972420145 | 2.912691431 | 3.829887332 | 4.721529277 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |



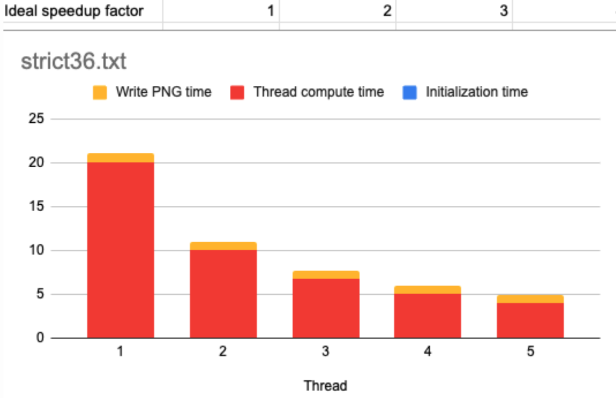



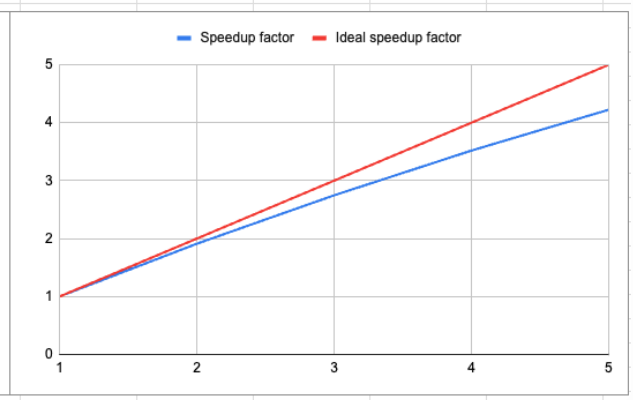

| Thread | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Initialization time | 0.000017 | 0.000024 | 0.000022 | 0.000022 | 0.000019 |
| Thread compute time | 46.753523 | 23.343529 | 15.578621 | 11.698381 | 9.357604 |
| Write PNG time | 1.000165 | 0.995231 | 1.001089 | 1.001179 | 1.002529 |
| Total time | 47.753705 | 24.338784 | 16.579732 | 12.699582 | 10.360152 |
| Speedup factor | 1 | 1.96204153 | 2.880245893 | 3.760258015 | 4.609363357 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |




| Thread | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Initialization time | 0.00002 | 0.000033 | 0.000019 | 0.000041 | 0.000021 |
| Thread compute time | 20.139368 | 10.070225 | 6.716372 | 5.036169 | 4.030731 |
| Write PNG time | 0.95577 | 0.96711 | 0.963097 | 0.958111 | 0.962724 |
| Total time | 21.095158 | 11.037368 | 7.679488 | 5.994321 | 4.993476 |
| Speedup factor | 1 | 1.91124895 | 2.746948494 | 3.519190581 | 4.224543785 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 |




以上幾張圖顯示初始化時間根本可以忽略。當每個thread的computation time很大時，write PNG time的佔比會非常小。

因此可以發現當總執行時間非常長時，我實作的平行化計算能夠逼近ideal speedup factor

也就是說如果撇除掉initialization time和write PNG time這兩個我沒做平行化的部分，其餘時間的平行化程度很高(可以從strict34.txt看出來)

## hybrid

經過pthread的實驗後，發現只要測strict34.txt就可以展現這隻程式的平行度優化結果(因為可平行化的部分佔比很大)

**Initialization time**用來計算初始化區域變數的時間

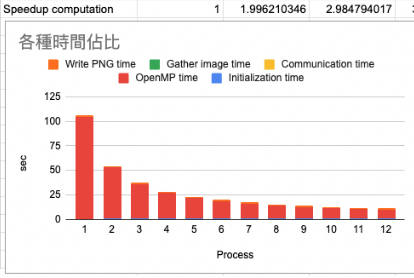
**OpenMP time**用來計算#pragma內的程式執行時間，由於這段會同時受process和thread數量影響，所以直接把這段當成是可平行化的最大部分

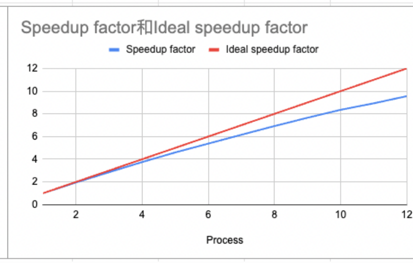
**Communication time**用來計算MPI_Gatherv所花時間

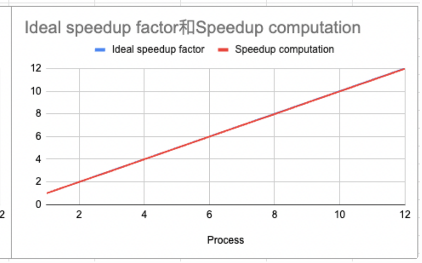
**Gather image time**用來計算把MPI_Gatherv組起來的圖片重組花的時間

**Write PNG time**用來計算最後畫圖的時間

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Node | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Process | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Thread | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Initialization time | 0.572333 | 0.742797 | 0.754974 | 0.656889 | 0.658948 | 0.778726 | 0.77498 | 0.690338 | 0.670201 | 0.682432 | 0.803367 | 0.812887 |
| OpenMP time | 104.159533 | 52.178636 | 34.896724 | 26.07274 | 20.862187 | 17.381102 | 14.893995 | 13.059597 | 11.589486 | 10.433455 | 9.488143 | 8.693935 |
| Communication time | 0.026706 | 0.02869 | 0.029594 | 0.036985 | 0.027696 | 0.030364 | 0.030447 | 0.029455 | 0.047198 | 0.031879 | 0.042589 | 0.033835 |
| Gather image time | 0.032135 | 0.027505 | 0.026384 | 0.029524 | 0.027305 | 0.032159 | 0.031954 | 0.026717 | 0.028126 | 0.027432 | 0.028417 | 0.030039 |
| Write PNG time | 1.552002 | 1.548266 | 1.537825 | 1.55069 | 1.547066 | 1.5514 | 1.539039 | 1.550449 | 1.55265 | 1.549093 | 1.549196 | 1.553284 |
| Total time | 106.342709 | 54.525894 | 37.245501 | 28.346828 | 23.123202 | 19.773751 | 17.270415 | 15.356556 | 13.887661 | 12.724291 | 11.911712 | 11.12398 |
| Speedup factor | 1 | 1.950315734 | 2.85518267 | 3.751485316 | 4.598961208 | 5.377973506 | 6.157507448 | 6.924906144 | 7.657352019 | 8.357456537 | 8.927575566 | 9.559771682 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Speedup computation | 1 | 1.996210346 | 2.984794017 | 3.994959218 | 4.992742755 | 5.992688668 | 6.993391162 | 7.975708056 | 8.987416094 | 9.983225403 | 10.9778629 | 11.98071219 |







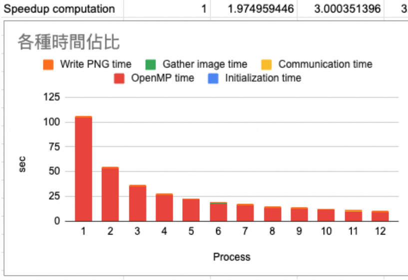

上圖是固定1個thread，實驗process從1~12個

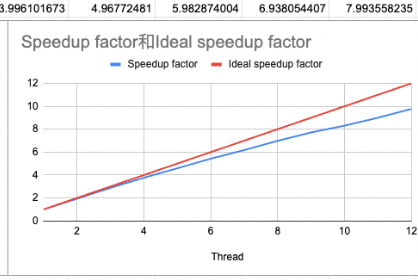
左圖是上述各種耗時的佔比圖

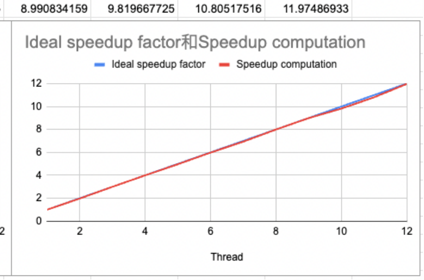
中間是總時間的speedup factor和ideal speedup factor的比較

右圖是如果不算無法平行的部分，也就是只算#pragma內的程式的speedup是否可以逼近ideal speedup factor

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Node | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Process | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Thread | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Initialization time | 0.572333 | 0.573443 | 0.57406 | 0.573721 | 0.572041 | 0.56985 | 0.58726 | 0.577853 | 0.581018 | 0.583119 | 0.571049 | 0.57586 |
| OpenMP time | 104.159533 | 52.740087 | 34.715778 | 26.065286 | 20.967251 | 17.409615 | 15.012787 | 13.030434 | 11.58508 | 10.607236 | 9.639782 | 8.698177 |
| Communication time | 0.026706 | 0.028174 | 0.033951 | 0.026475 | 0.026754 | 0.032457 | 0.026429 | 0.031341 | 0.030468 | 0.026405 | 0.031744 | 0.032265 |
| Gather image time | 0.032135 | 0.025645 | 0.032495 | 0.031784 | 0.031799 | 0.028885 | 0.031998 | 0.031327 | 0.03176 | 0.026511 | 0.031411 | 0.028462 |
| Write PNG time | 1.552002 | 1.555098 | 1.539508 | 1.549659 | 1.567844 | 1.551151 | 1.555772 | 1.550404 | 1.554634 | 1.555677 | 1.541575 | 1.550388 |
| Total time | 106.342709 | 54.922447 | 36.895792 | 28.246925 | 23.165689 | 19.591958 | 17.214246 | 15.221359 | 13.78296 | 12.798948 | 11.815561 | 10.885152 |
| Speedup factor | 1 | 1.936233996 | 2.882244918 | 3.764753473 | 4.590526489 | 5.427875509 | 6.177599007 | 6.986413565 | 7.715520396 | 8.308707012 | 9.000225127 | 9.769519893 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Speedup computation | 1 | 1.974959446 | 3.000351396 | 3.996101673 | 4.96772481 | 5.982874004 | 6.938054407 | 7.993558235 | 8.990834159 | 9.819667725 | 10.80517516 | 11.97486933 |

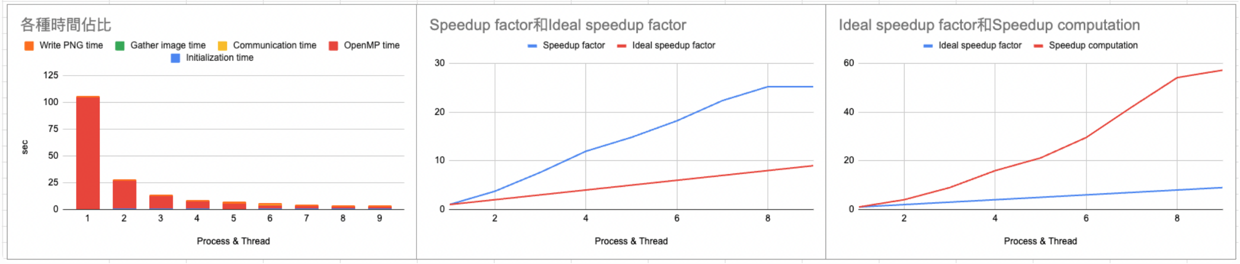







上圖是固定1個process，實驗thread從1~12個

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Node | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Process | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Thread | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Initialization time | 0.572333 | 0.738736 | 0.743429 | 0.749533 | 0.642292 | 0.694511 | 0.673534 | 0.672644 | 0.791758 |
| OpenMP time | 104.159533 | 26.036886 | 11.58246 | 6.524702 | 4.911636 | 3.519272 | 2.472512 | 1.920887 | 1.817881 |
| Communication time | 0.026706 | 0.060347 | 0.028648 | 0.030443 | 0.030438 | 0.038828 | 0.034927 | 0.039373 | 0.033062 |
| Gather image time | 0.032135 | 0.027638 | 0.029302 | 0.025931 | 0.030771 | 0.032217 | 0.023757 | 0.028127 | 0.024868 |
| Write PNG time | 1.552002 | 1.553794 | 1.546901 | 1.553506 | 1.557579 | 1.54946 | 1.54595 | 1.550924 | 1.549311 |
| Total time | 106.342709 | 28.417401 | 13.93074 | 8.884115 | 7.172716 | 5.834288 | 4.75068 | 4.211955 | 4.21688 |
| Speedup factor | 1 | 3.742168716 | 7.633672655 | 11.96998339 | 14.82600301 | 18.22719567 | 22.38473419 | 25.24782648 | 25.21833891 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Speedup computation | 1 | 4.000460462 | 8.992867923 | 15.9638759 | 21.20668816 | 29.5968976 | 42.12700808 | 54.22470609 | 57.29722298 |




上圖是同時增加process和thread的數量看看speedup factor是否能呈指數成長
實驗結果大概在5個process和5個thread時就上不去了

## Profile

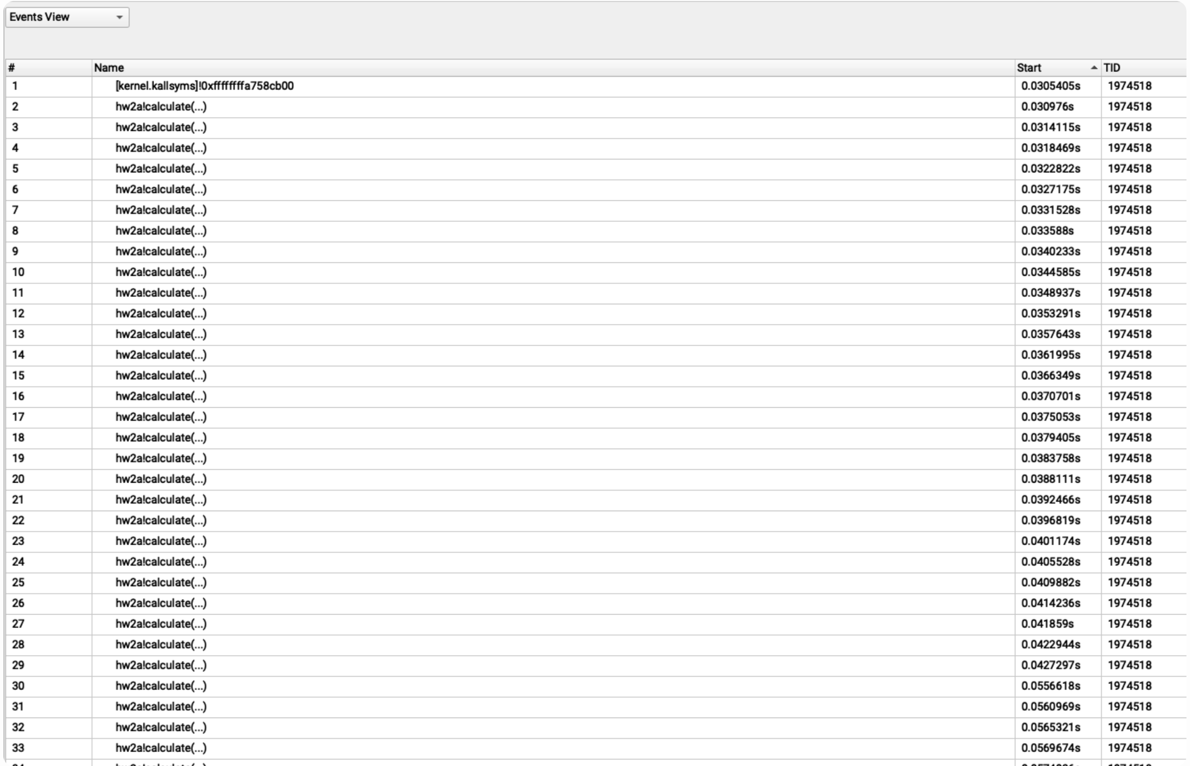
### pthread


從profiler可以看出幾個點：

1. 有些thread中間會有一小段空白(約8ms)，這些應該是做完工作準備拿新工作時，別的thread已經搶到lock，所以是等待lock的時間。
2. 有時候thread在等lock，原本在使用的CPU core會被其他thread搶佔，所以等拿到lock後，會交換執行的CPU core。

Events View

| # | Name | Start | Duration | TID |
|---|---|---|---|---|
| 6 | pthread_join | 0.0305221s | 21.387 s | 1974503 |
| 4372 | fclose | 22.9377s | 24.129 ms | 1974503 |
| 8 | fopen | 21.4182s | 5.495 ms | 1974503 |
| 7 | pthread_join | 21.4177s | 484.412 μs | 1974503 |
| 4 | pthread_create | 0.0302433s | 169.291 μs | 1974503 |
| 1 | pthread_create | 0.0297675s | 158.784 μs | 1974503 |
| 2 | pthread_create | 0.0299278s | 139.272 μs | 1974503 |
| 3 | pthread_create | 0.0300952s | 134.617 μs | 1974503 |
| 5 | pthread_create | 0.0304147s | 101.582 μs | 1974503 |
| 2429 | fwrite | 22.2721s | 12.026 μs | 1974503 |
| 2114 | fwrite | 22.1634s | 11.742 μs | 1974503 |
| 2146 | fwrite | 22.1739s | 11.699 μs | 1974503 |
| 3217 | fwrite | 22.5435s | 11.250 μs | 1974503 |
| 350 | fwrite | 21.5423s | 11.069 μs | 1974503 |
| 10 | fwrite | 21.4248s | 11.062 μs | 1974503 |
| 1705 | fwrite | 22.019s | 10.906 μs | 1974503 |
| 791 | fwrite | 21.6957s | 10.905 μs | 1974503 |
| 2902 | fwrite | 22.4352s | 10.838 μs | 1974503 |
| 1642 | fwrite | 21.9966s | 10.824 μs | 1974503 |
| 476 | fwrite | 21.5862s | 10.750 μs | 1974503 |
| 2744 | fwrite | 22.3801s | 10.726 μs | 1974503 |
| 917 | fwrite | 21.7393s | 10.706 μs | 1974503 |
| 4131 | fwrite | 22.856s | 10.690 μs | 1974503 |
| 382 | fwrite | 21.5536s | 10.679 μs | 1974503 |
| 1012 | fwrite | 21.7721s | 10.678 μs | 1974503 |
| 2587 | fwrite | 22.3258s | 10.652 μs | 1974503 |

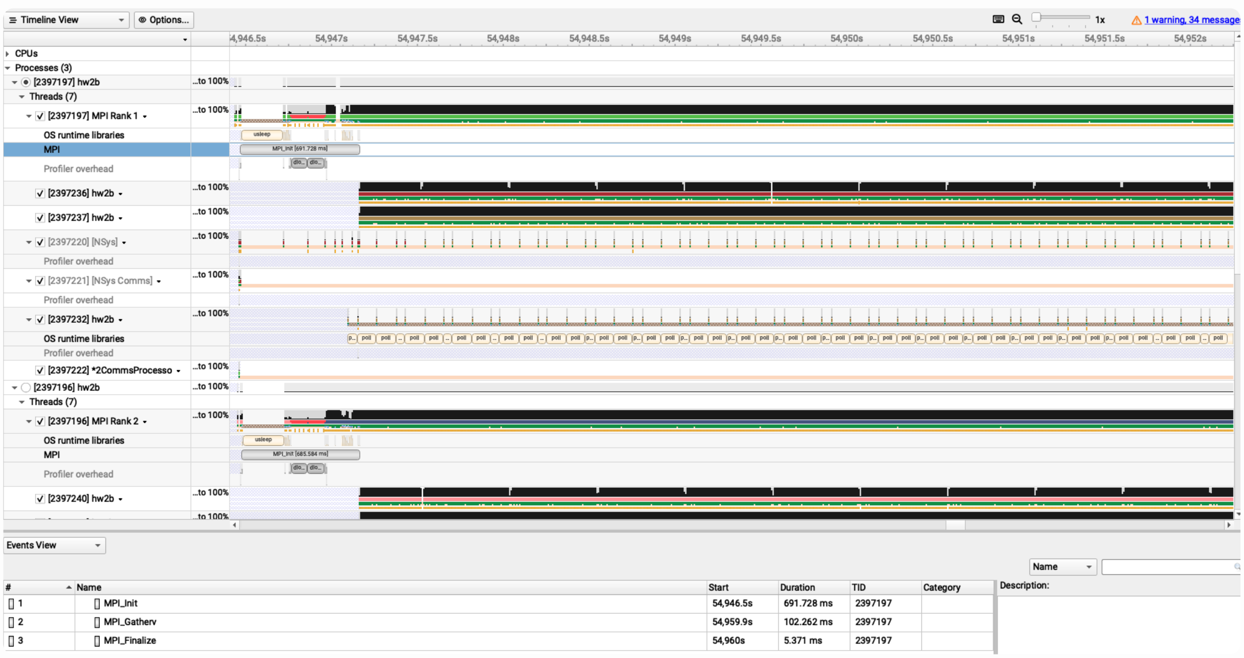


Events View

| # | Name | Start | TID |
|---|---|---|---|
| 1 | [kernel.kallsyms]!0xffffffffa758cb00 | 0.0305405s | 1974518 |
| 2 | hw2a!calculate(...) | 0.030976s | 1974518 |
| 3 | hw2a!calculate(...) | 0.0314115s | 1974518 |
| 4 | hw2a!calculate(...) | 0.0318469s | 1974518 |
| 5 | hw2a!calculate(...) | 0.0322822s | 1974518 |
| 6 | hw2a!calculate(...) | 0.0327175s | 1974518 |
| 7 | hw2a!calculate(...) | 0.0331528s | 1974518 |
| 8 | hw2a!calculate(...) | 0.033588s | 1974518 |
| 9 | hw2a!calculate(...) | 0.0340233s | 1974518 |
| 10 | hw2a!calculate(...) | 0.0344585s | 1974518 |
| 11 | hw2a!calculate(...) | 0.0348937s | 1974518 |
| 12 | hw2a!calculate(...) | 0.0353291s | 1974518 |
| 13 | hw2a!calculate(...) | 0.0357643s | 1974518 |
| 14 | hw2a!calculate(...) | 0.0361995s | 1974518 |
| 15 | hw2a!calculate(...) | 0.0366349s | 1974518 |
| 16 | hw2a!calculate(...) | 0.0370701s | 1974518 |
| 17 | hw2a!calculate(...) | 0.0375053s | 1974518 |
| 18 | hw2a!calculate(...) | 0.0379405s | 1974518 |
| 19 | hw2a!calculate(...) | 0.0383758s | 1974518 |
| 20 | hw2a!calculate(...) | 0.0388111s | 1974518 |
| 21 | hw2a!calculate(...) | 0.0392466s | 1974518 |
| 22 | hw2a!calculate(...) | 0.0396819s | 1974518 |
| 23 | hw2a!calculate(...) | 0.0401174s | 1974518 |
| 24 | hw2a!calculate(...) | 0.0405528s | 1974518 |
| 25 | hw2a!calculate(...) | 0.0409882s | 1974518 |
| 26 | hw2a!calculate(...) | 0.0414236s | 1974518 |
| 27 | hw2a!calculate(...) | 0.041859s | 1974518 |
| 28 | hw2a!calculate(...) | 0.0422944s | 1974518 |
| 29 | hw2a!calculate(...) | 0.0427297s | 1974518 |
| 30 | hw2a!calculate(...) | 0.0556618s | 1974518 |
| 31 | hw2a!calculate(...) | 0.0560969s | 1974518 |
| 32 | hw2a!calculate(...) | 0.0565321s | 1974518 |
| 33 | hw2a!calculate(...) | 0.0569674s | 1974518 |

每個thread做完一次工作大概是0.0004352秒

### hybrid




| # | Name | Start | Duration | TID | Category |
|---|---|---|---|---|---|
| 1 | MPI_Init | 54,946.5s | 691.728 ms | 2397197 | |
| 2 | MPI_Gatherv | 54,959.9s | 102.262 ms | 2397197 | |
| 3 | MPI_Finalize | 54,960s | 5.371 ms | 2397197 | |

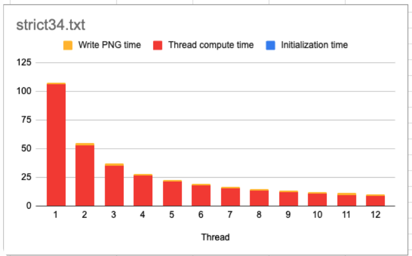
Hybrid方法就不像pthread會用lock去管理工作，而是在MPI_Init後就針對不同rank的process分配好需要做的工作，因此不會有中間的小斷點

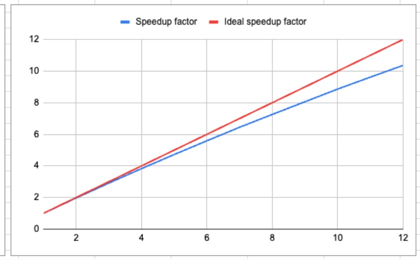
## Discussion

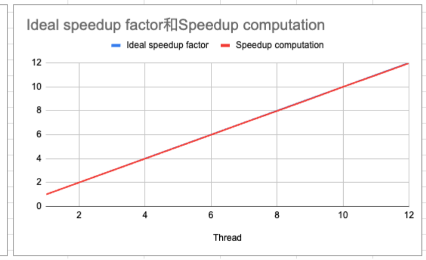
### Scalability

拿strict34.txt當作input來計算從1個thread scale up到12個thread。

| Thread | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Initialization time | 0.000019 | 0.000018 | 0.000018 | 0.000019 | 0.00002 | 0.000017 | 0.00002 | 0.000018 | 0.000019 | 0.000023 | 0.000022 | 0.000019 |
| Thread compute time | 106.299448 | 53.135242 | 35.483878 | 26.625018 | 21.30085 | 17.751017 | 15.196399 | 13.316827 | 11.83831 | 10.631568 | 9.68083 | 8.872913 |
| Write PNG time | 1.541206 | 1.539031 | 1.540511 | 1.532625 | 1.53933 | 1.541227 | 1.532981 | 1.540735 | 1.538777 | 1.536085 | 1.530049 | 1.53586 |
| Total time | 107.840673 | 54.674291 | 37.024407 | 28.157662 | 22.8402 | 19.292261 | 16.7294 | 14.85758 | 13.377106 | 12.167676 | 11.210901 | 10.408792 |
| Speedup factor | 1 | 1.972420145 | 2.912691431 | 3.829887332 | 4.721529277 | 5.589841077 | 6.44617697 | 7.258293275 | 8.061584696 | 8.862881704 | 9.619269049 | 10.36053684 |
| Ideal speedup factor | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Speedup computation | 1 | 2.0005451 | 2.995711123 | 3.992464831 | 4.990385266 | 5.988358188 | 6.995041917 | 7.982340538 | 8.979275589 | 9.998473226 | 10.98040643 | 11.9802198 |



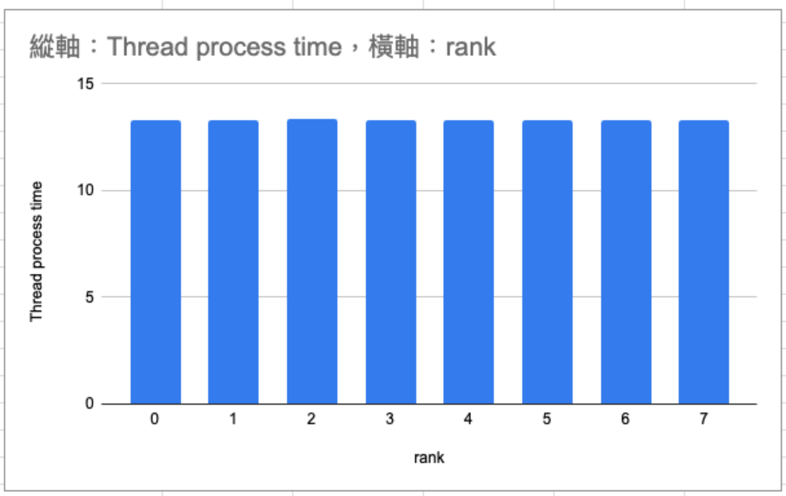




實驗結果發現當thread數量逐漸scale up，speedup factor逐漸趨緩，看起來會收斂到一個最終值
但是我又另外做了一個實驗，只計算可平行化的部分speedup會不會也收斂
結果發現當process在12個以下時，沒辦法看出明顯得收斂，甚至接近ideal speedup factor
因此可以得到一個結論，撇除可平行化的部分，剩餘不可平行化的部分就是無法逼近ideal speedup factor的bottleneck

**Load balancing**

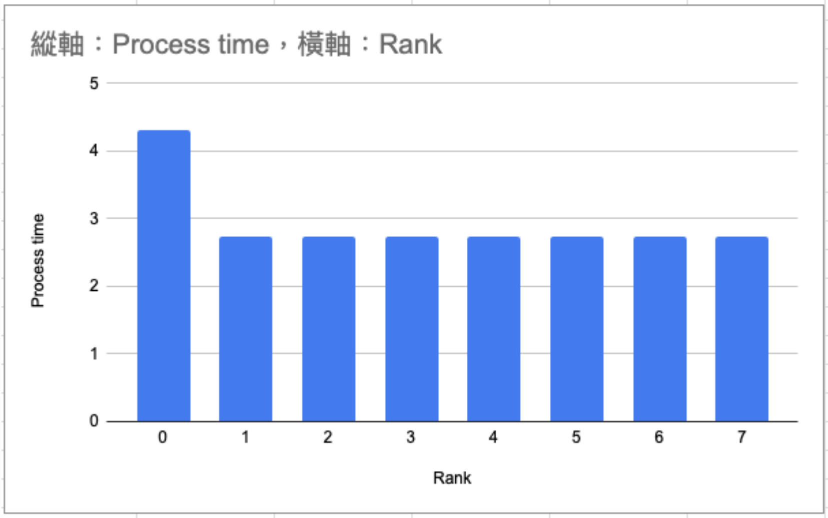
拿strict34.txt當作input來實驗8個thread的執行時間

| rank | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Thread process time | 13.327886 | 13.328211 | 13.328577 | 13.323418 | 13.325058 | 13.325827 | 13.325373 | 13.32304 |




**Pthread**實驗結果顯示，每個thread執行時間很平均，算是達成一個不錯的load balance
藉由控制blockPerThread來達到更fine-grained的工作分配是可行的
然而我做了許多小實驗，從blockPerThread = 30，做到blockPerThread = 2500，跑judge的總時間是漸減的。由此可知，當工作被切得越小份時，雖然能增加計算資源的利用，但是每次重新分配資源到vector所產生的overhead也會相對提升。因此當圖片越大時，blockPerThread就要相對上升，以減少overhead的產生

| Rank | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Process time | 4.309457 | 2.744195 | 2.745263 | 2.745033 | 2.744374 | 2.744251 | 2.744965 | 2.744752 |




**Hybrid**方法因為rank0要負責做圖片的aggregate，所以會花比較多時間，因此會成為效能瓶頸
其餘的process執行時間都差不多，算是達成不錯的load balance
以目前簡單做的小實驗來看，data parallelism的schedule用static、branch = 10效果最好

## Conclusion

經過這次作業，我了解到不是每個程式都適合拿來做平行計算優化。這次作業的程式就很適合用平行計算優化。從sequential的程式來看，每個計算工作都互相獨立，且性質相同，非常適合同時做計算。不論是每個工作的計算能平行，還是最後畫圖的部分，我相信都是可以用平行計算去逼近ideal speedup的。
實作上遇到比較大的困難大概是看不到vector裡的值，所以debug的時候會需要把vector裡的所有元素都倒出來，會需要看一堆log來檢查每個工作是不是都做了，也需要檢查每個工作是不是在正確的時間結束工作。