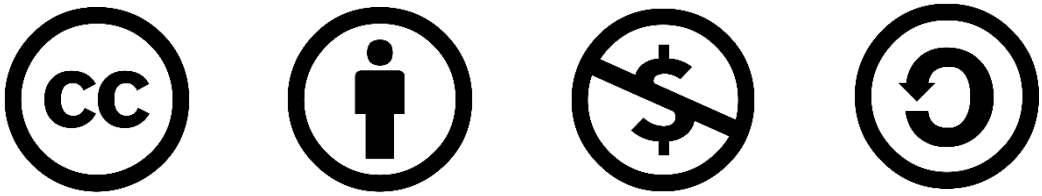


Report Common Assignment 1

Graph traversal:
Breath First Search Algorithm

Lecturer: Francesco Moscato - fmoscato@unisa.it

Student: Canzolino Gianluca - 0622701806 - g.canzolino3@studenti.unisa.it

Sommario

Breath First Search

In questo documento viene trattata la parallelizzazione e la valutazione delle performance dell'algoritmo "Breath First Search" utilizzando OpenMP.

Setup sperimentale

Hardware

CPU

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 113
model name      : AMD Ryzen 7 3700X 8-Core Processor
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 3599.998
cache size      : 512 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall
nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good nopl
tsc_reliable nonstop_tsc cpuid extd_apicid pni pclmulqdq ssse3
fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand
hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignsse
3dnowprefetch osvw topoext ssbd ibpb stibp vmmcall fsgsbase bmi1
avx2 smep bmi2 rdseed adx smap clflushopt clwb sha_ni xsaveopt
xsavec xgetbv1 xsaves clzero xsaveerptr virt_ssbd arat umip rdpid
bugs                    :  sysret_ss_attrs  spectre_v1  spectre_v2
spec_store_bypass
bogomips        : 7199.99
TLB size        : 3072 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management:
```

RAM

```
MemTotal:       26202916 kB
MemFree:        25992316 kB
MemAvailable:   25807988 kB
Buffers:            7092 kB
Cached:            49904 kB
SwapCached:            0 kB
Active:            46532 kB
Inactive:          14048 kB
Active(anon):         76 kB
Inactive(anon):     3940 kB
Active(file):      46456 kB
Inactive(file):    10108 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       7340032 kB
SwapFree:        7340032 kB
Dirty:                60 kB
Writeback:             0 kB
AnonPages:          3948 kB
Mapped:             4160 kB
Shmem:                68 kB
KReclaimable:      18076 kB
Slab:              50452 kB
SReclaimable:      18076 kB
SUnreclaim:        32376 kB
KernelStack:        3168 kB
PageTables:          304 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    20441488 kB
Committed_AS:       6540 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       24688 kB
VmallocChunk:          0 kB
Percpu:             5120 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
FileHugePages:         0 kB
FilePmdMapped:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
DirectMap4k:       17408 kB
DirectMap2M:     3686400 kB
DirectMap1G:    23068672 kB
```

Software

Python 3.8.10

GCC 9.3.0

WLS 2 on Windows 10

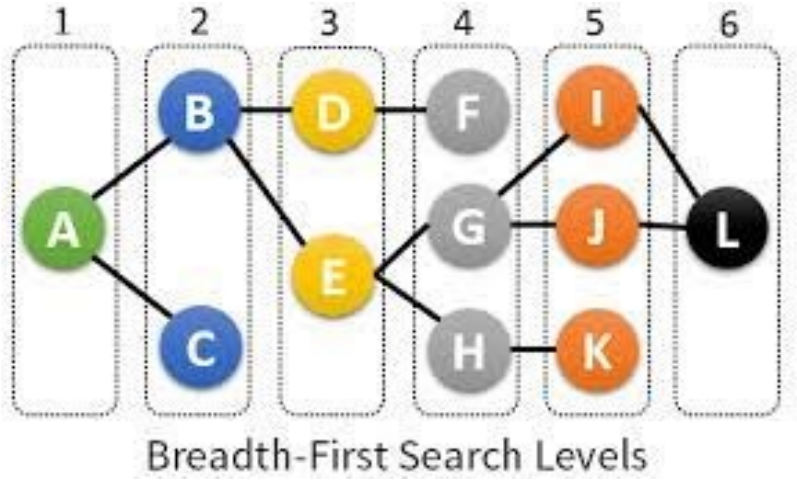
Riguardo le misure

Ogni misurazione è stata eseguita 50 volte in modo tale da avere un'accuratezza migliore. Tutti i test sono stati eseguiti con le varie ottimizzazioni (O0, O1, O2, O3) su 4 grafi differenti, con il rispettivo numero di vertici: 1000, 10000, 100000, 300000.

Report Breath-First Search

Breve descrizione

Nella teoria dei grafi, la Breadth-First Search (in italiano “ricerca in ampiezza”), è un algoritmo di ricerca per grafi che partendo da un vertice (o nodo) detto sorgente permette di cercare tutti gli altri nodi. La BFS si basa su livelli, ovvero per quanti nodi separano il nodo sorgente da un generico nodo v.



Algortimo Sequenziale

Input: un grafo *G* e un nodo radice *v* appartenente a *G*

```
1   function BFS(G,v):
2       crea una coda Q
3       inserisci v in Q
4       marca v
5       while Q non è vuota:
6           t ← Q.dequeue()
7           for all archi e in G.incident_edge(t) do
8              u ← G.opposite_node(t,e)
9              if u non è marcato:
10                   marca u
11                   inserisci u in Q
12      return none
```

L’algoritmo può essere suddiviso in due semplici passi:

1) Prendo il vertice v dalla coda Q
2) Aggiungo in coda tutti i vicini di v che non sono già stati marcati

Implementazione algoritmo sequenziale

```
void bfs_naive(struct Graph* graph, struct queue* bfs_queue) {
  struct queue* q = createQueue(graph->numVertices);
  int adjVertex;
  int currentVertex;
  struct node* v;

  graph->visited[START_NODE] = 1;
  enqueue(q, START_NODE);
  enqueue(bfs_queue, START_NODE);

  while (!isEmpty(q)) {
    currentVertex = dequeue(q);

    v = graph->adjLists[currentVertex];

    for(int i=0; i<v->n_neighbours; i++){
      adjVertex = v->neighbours[i]->vertex;
      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
        enqueue(bfs_queue, adjVertex);
      }
    }
  }
}
```

Punti di forza

Il punto di forza più evidente è la semplicità. Essendo un algoritmo che utilizza un ciclo for all'interno, è adatto alla parallelizzazione in modo molto efficiente su un singolo processore multicore. È necessario però che il grafo e la coda, essendo condivisi, siano in una regione critica per evitare incongruenze.

Punti di debolezza

Il punto di debolezza principale sta nell'algoritmo della BFS. Se un grafo con N vertici ha N livelli, il parallelismo non avverrebbe.

Implementazione e dettagli specifici

```
void bfs_parallel(struct Graph* graph) {
  int next_queue, current_queue = 0;
  int l[2];
  l[0] = 1;
  int q[2][SIZE];
  int level = 2;

  graph->visited[START_NODE] = 1;
  q[current_queue][START_NODE] = START_NODE;

  while(1){
    next_queue = (current_queue + 1) % 2;
    l[next_queue] = 0;
    #pragma omp parallel shared(graph, q, l)
    {
      int currentVertex;
      struct node* v;
      #pragma omp for
      for(int i=0; i<l[current_queue]; i++){
        currentVertex = q[current_queue][i];
        v = graph->adjLists[currentVertex];
        for(int j=0; j<v->n_neighbours; j++){
          if (graph->visited[v->neighbours[j]->vertex] == 0) {
            #pragma omp critical
            {
              graph->visited[v->neighbours[j]->vertex] = level;
              q[next_queue][l[next_queue]++] = v->neighbours[j]->vertex;
            }
          }
        }
      }
    }
    if(l[next_queue] == 0){
      break;
    }
    else{
      current_queue=next_queue;
      level++;
    }
  }
}
```

La prima modifica sta nel fatto di non creare una vera e propria coda, dato che essa non è sequenziale (è possibile rimuovere un elemento alla volta), ma di creare un array ***q***.

Ogni livello cambia q, in modo tale da avere due “code” separate. Questo è stato possibile istanziando q come un array bidimensionale.

```
#pragma omp parallel shared(graph, q, l)
{
  int currentVertex;
  struct node* v;
  #pragma omp for
  for(int i=0; i<l[current_queue]; i++){
    currentVertex = q[current_queue][i];
    v = graph->adjLists[currentVertex];
    for(int j=0; j<v->n_neighbours; j++){
      if (graph->visited[v->neighbours[j]->vertex] == 0) {
        #pragma omp critical
        {
          graph->visited[v->neighbours[j]->vertex] = level;
          q[next_queue][l[next_queue]++] = v->neighbours[j]->vertex;
        }
      }
    }
  }
}
```

Come già accennato in precedenza, la parallelizzazione è semplice da implementare tramite l'uso di un *pragma parallel for*. È importante sottolineare che il ***grafo***, la ***coda*** e ***l*** (che rappresenta il numero di nodi nel livello) sono elementi condivisi tra i vari thread. Infatti, è stata utilizzata la clausola *shared* per etichettarli come condivisi.

L'algoritmo è simile al sequenziale con l'unica differenza che i vicini vengono esplorati in parallelo. Ogni thread prende in considerazione un vertice ed esplora i propri vicini. Se il numero di elementi in coda è piccolo (grafo poco denso), ci saranno più thread che nodi da esplorare diminuendo l'efficienza.

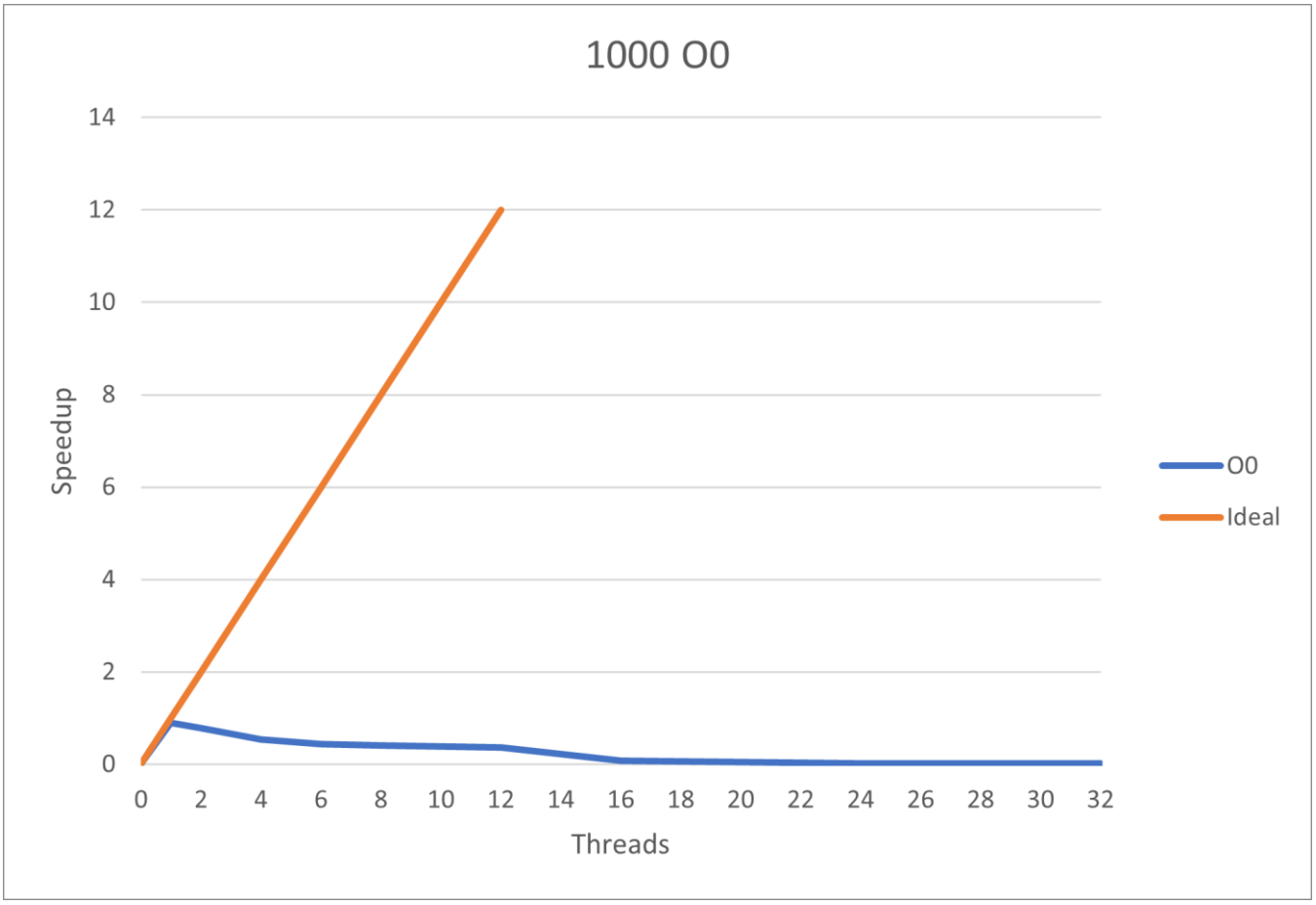
Per ogni vicino di ogni nodo nella coda viene controllato che non sia stato già visitato, nel caso non lo fosse, viene marcato come visitato (con valore pari al livello) e viene aggiunto in coda, la quale sarà visitata l'iterazione successiva.

Quest'ultima operazione è molto delicata, dato che potrebbe accadere che due thread stiano trattando due vertici diversi ma che hanno come vicino lo stesso nodo. Per evitare che il nodo venga aggiunto più volte in coda e per evitare sovrascritture pericolose, è necessario utilizzare un *pragma omp critical* per evitare questi scenari.

Analisi delle misure

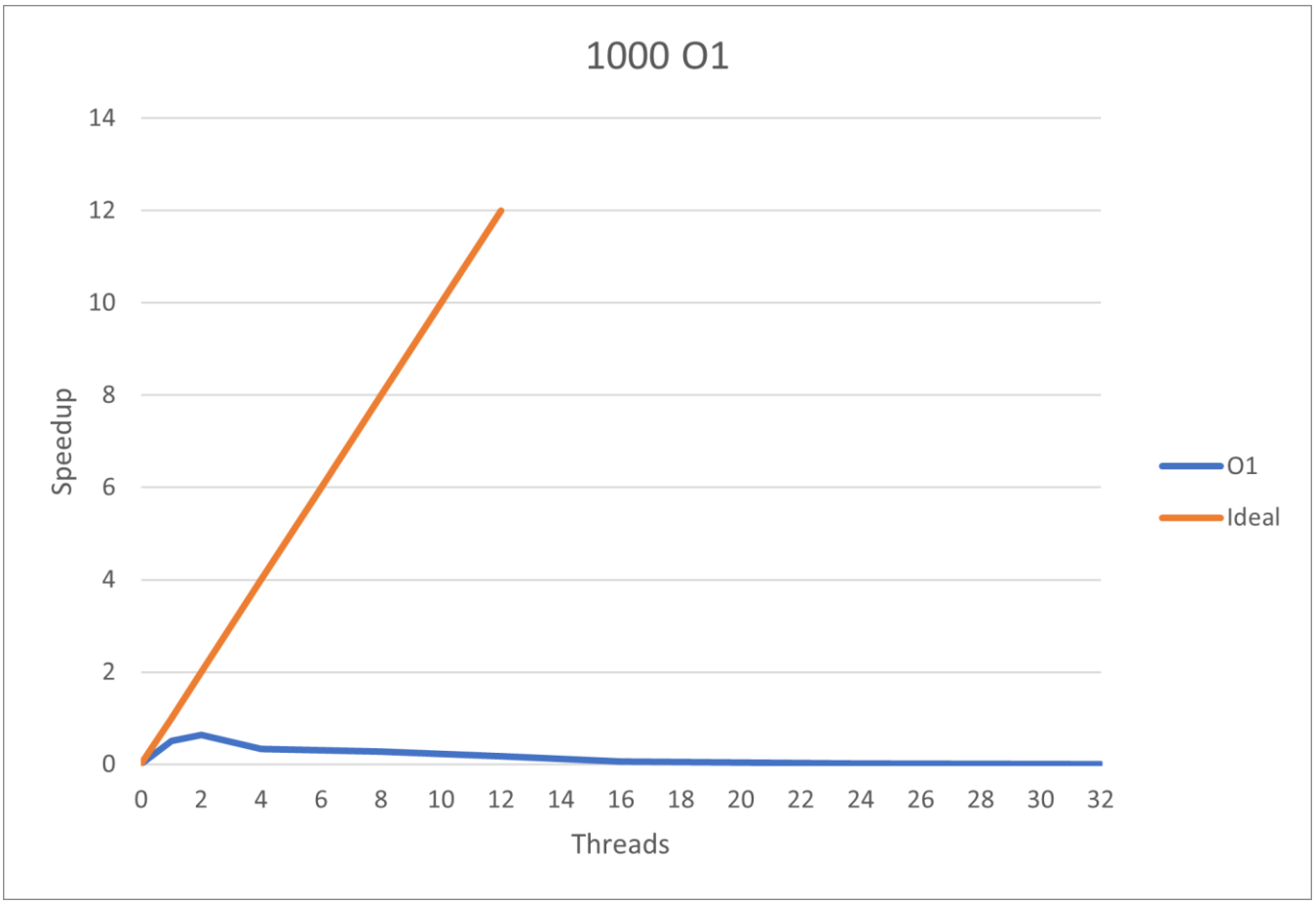
Misure O0-1000

Type	Time	Speedup
Sequential	9,84E-05	/
Threads 1	0,00011	0,897955
Threads 2	0,000127	0,777216
Threads 4	0,000181	0,543176
Threads 6	0,000222	0,442277
Threads 8	0,000238	0,413674
Threads 12	0,000273	0,360446
Threads 16	0,001297	0,075877
Threads 24	0,004262	0,023084
Threads 32	0,005673	0,017343



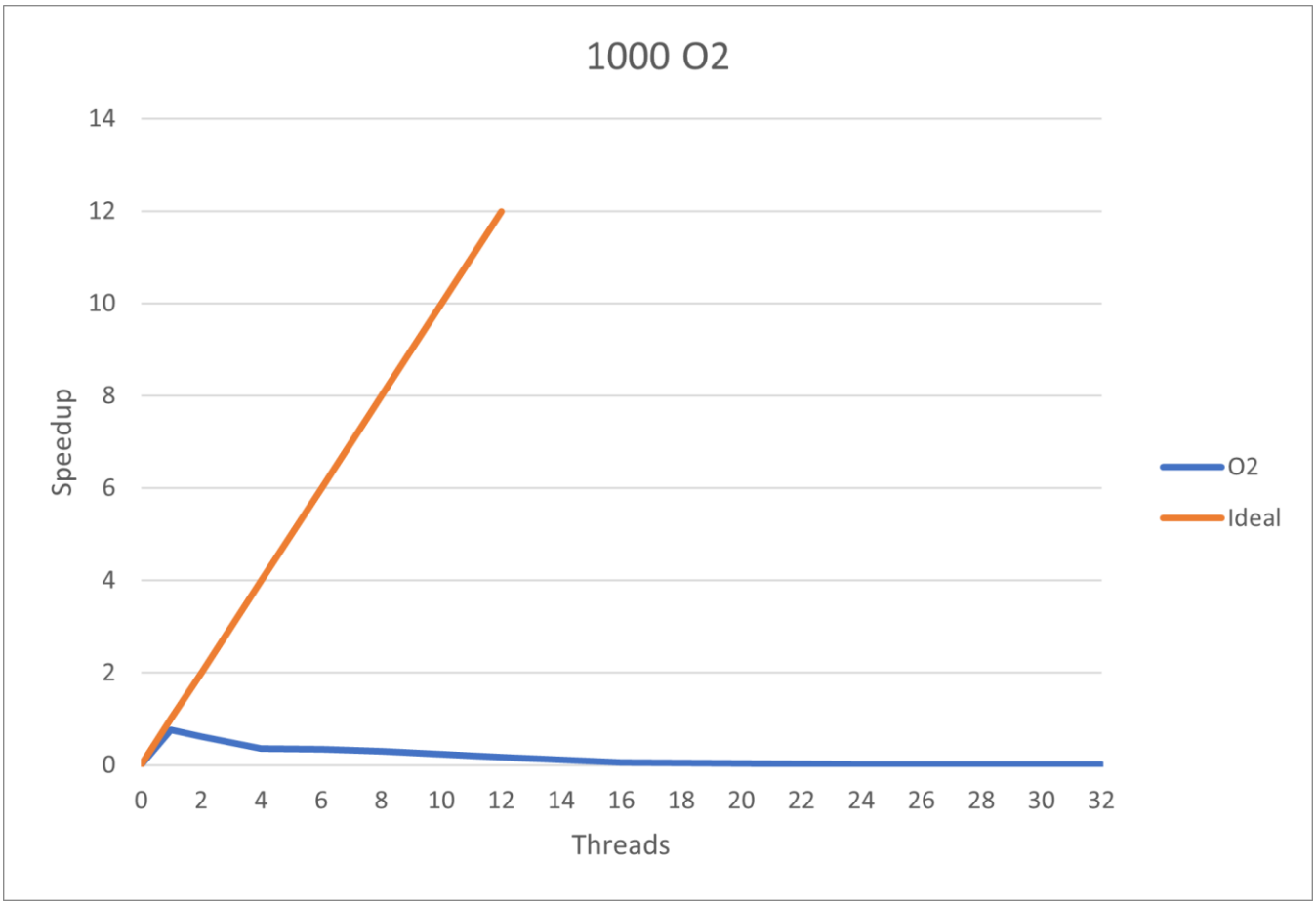
Misure O1-1000

Type	Time	Speedup
Sequential	*5,56E-05*	*/*
Threads 1	*0,000109*	*0,509068*
Threads 2	*8,73E-05*	*0,636801*
Threads 4	*0,000167*	*0,333054*
Threads 6	*0,000178*	*0,311407*
Threads 8	*0,000197*	*0,282304*
Threads 12	*0,000323*	*0,17183*
Threads 16	*0,000956*	*0,058122*
Threads 24	*0,004199*	*0,013238*
Threads 32	*0,005667*	*0,009808*



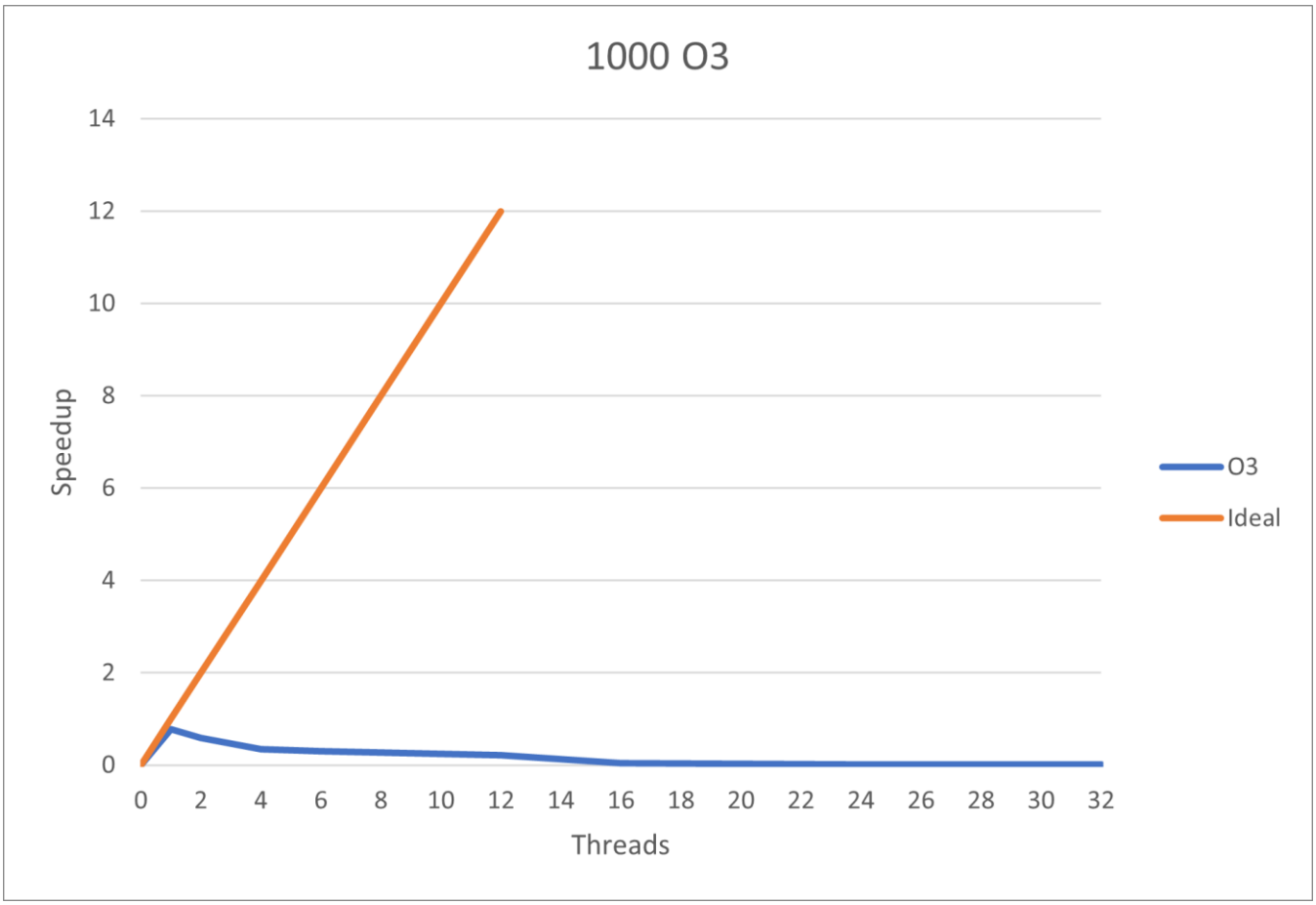
Misure O2-1000

Type	Time	Speedup
Sequential	*6,05E-05*	*/*
Threads 1	*7,92E-05*	*0,764141*
Threads 2	*9,82E-05*	*0,616168*
Threads 4	*0,000166*	*0,363702*
Threads 6	*0,000179*	*0,338063*
Threads 8	*0,000205*	*0,295421*
Threads 12	*0,000353*	*0,171289*
Threads 16	*0,000972*	*0,06229*
Threads 24	*0,00419*	*0,014445*
Threads 32	*0,005643*	*0,010725*



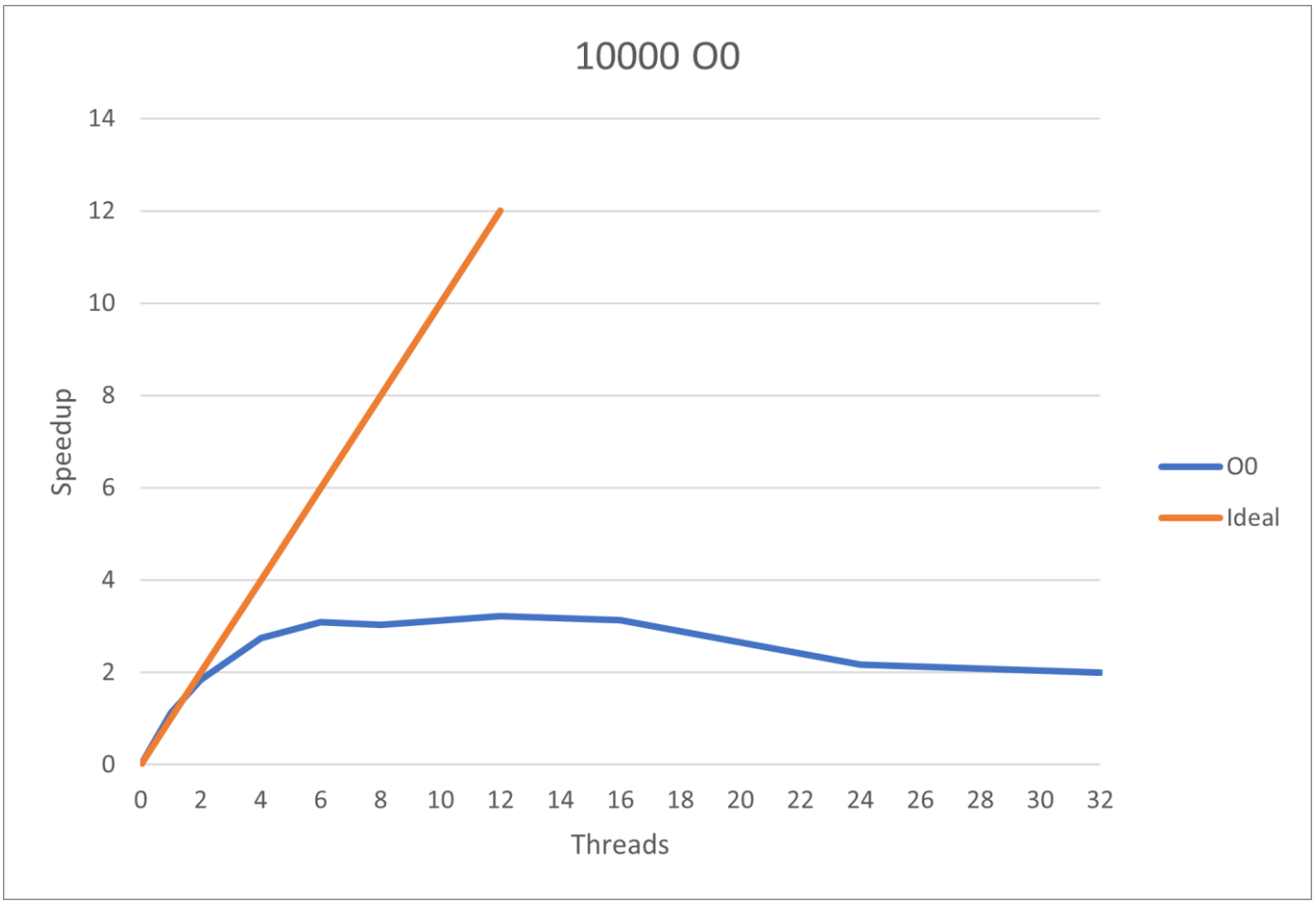
Misure O3-1000

Type	Time	Speedup
Sequential	*5,55E-05*	*/*
Threads 1	*7,19E-05*	*0,772613*
Threads 2	*9,36E-05*	*0,593416*
Threads 4	*0,000159*	*0,350019*
Threads 6	*0,000185*	*0,30027*
Threads 8	*0,000199*	*0,278519*
Threads 12	*0,000262*	*0,211925*
Threads 16	*0,001296*	*0,042835*
Threads 24	*0,004228*	*0,01313*
Threads 32	*0,005864*	*0,009468*



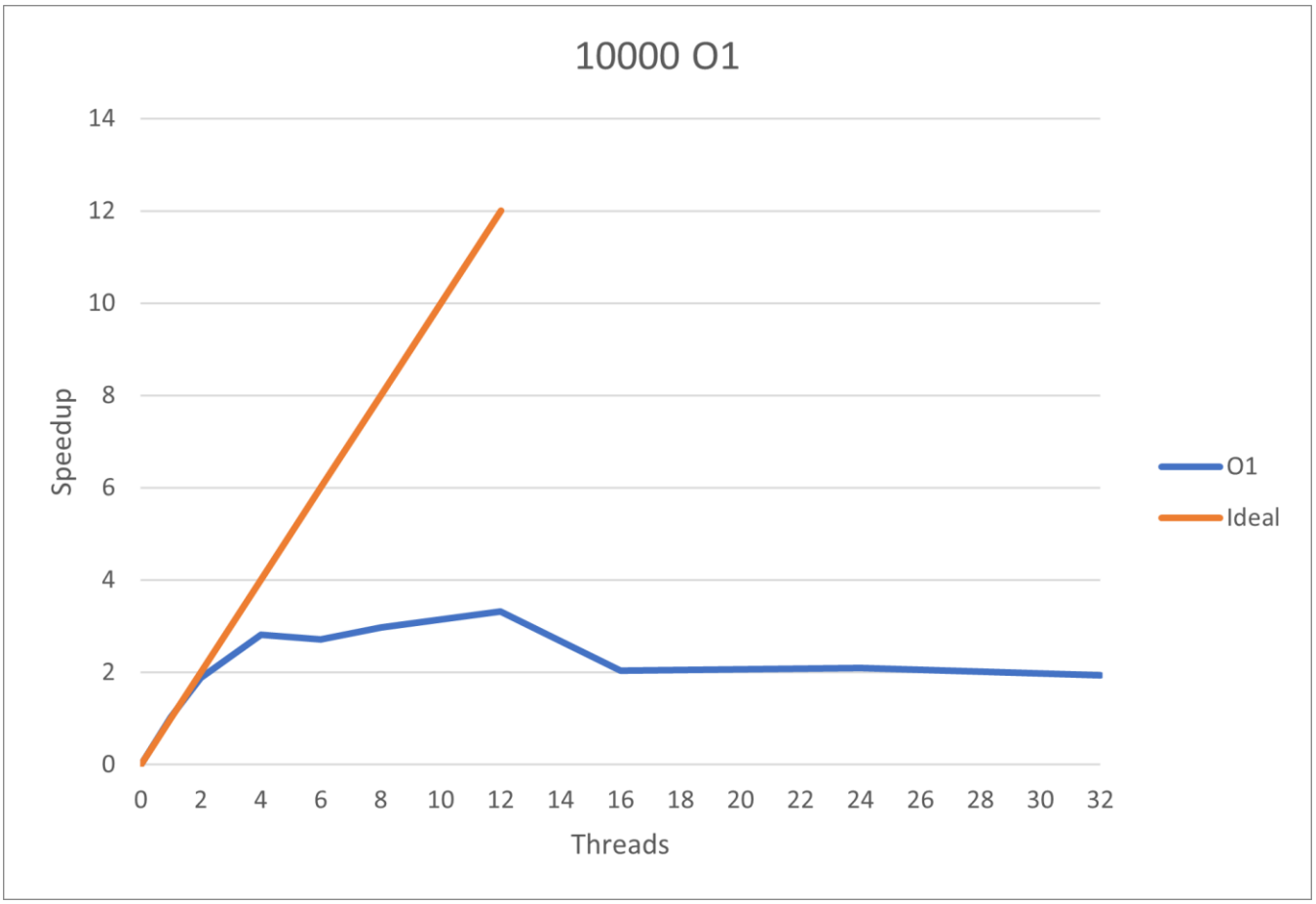
Misure O0-10000

Type	Time	Speedup
Sequential	*0,018199*	*/*
Threads 1	*0,016318*	*1,115246*
Threads 2	*0,009874*	*1,84301*
Threads 4	*0,00663*	*2,744718*
Threads 6	*0,005893*	*3,088337*
Threads 8	*0,005996*	*3,035373*
Threads 12	*0,005659*	*3,21607*
Threads 16	*0,005798*	*3,13866*
Threads 24	*0,008406*	*2,164869*
Threads 32	*0,00913*	*1,993356*



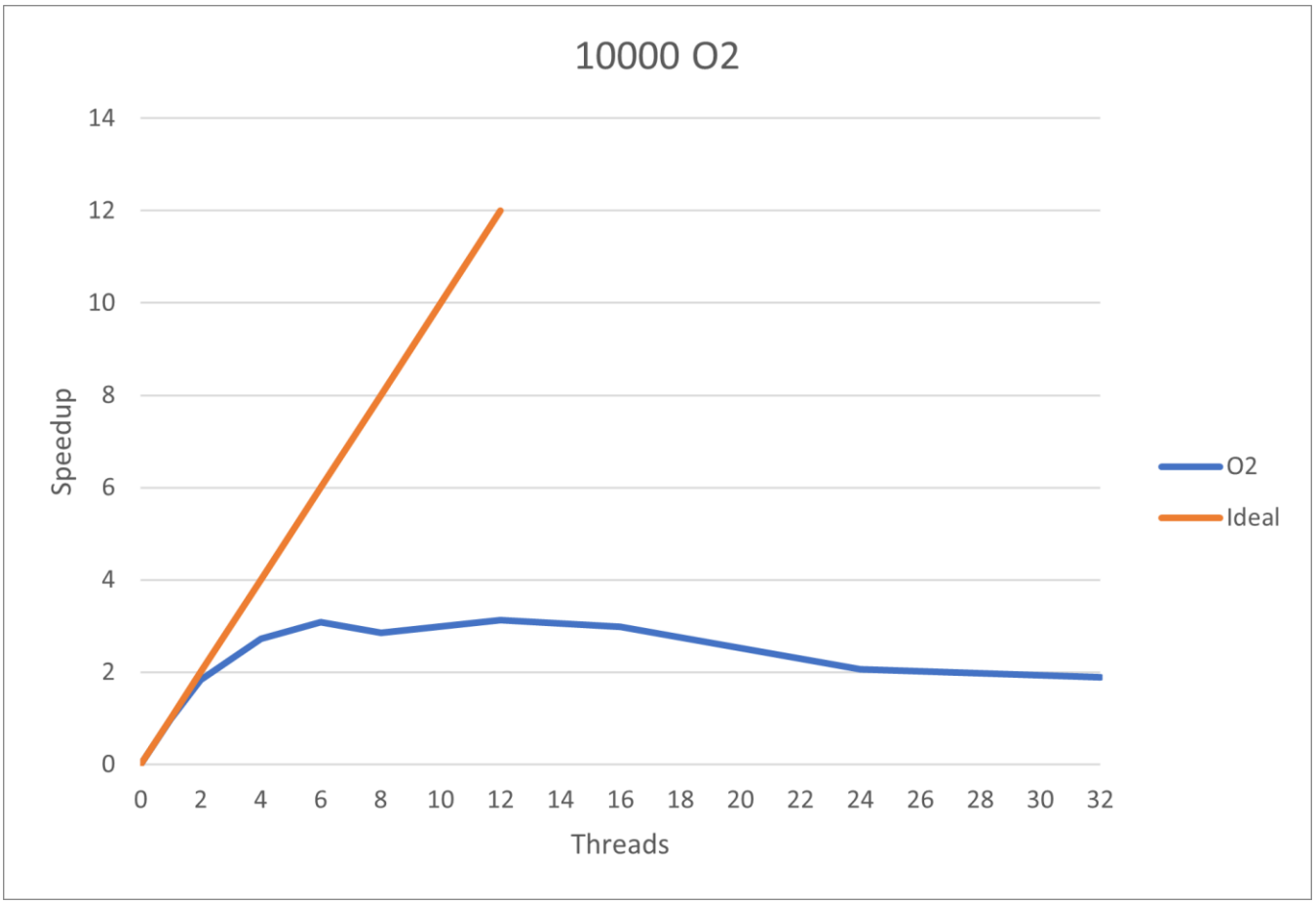
Misure O1-10000

Type	Time	Speedup
Sequential	*0,016972*	*/*
Threads 1	*0,01648*	*1,029816*
Threads 2	*0,009041*	*1,87715*
Threads 4	*0,006036*	*2,811878*
Threads 6	*0,006251*	*2,714995*
Threads 8	*0,005711*	*2,971988*
Threads 12	*0,005111*	*3,320409*
Threads 16	*0,008305*	*2,043538*
Threads 24	*0,008108*	*2,093288*
Threads 32	*0,008747*	*1,940217*



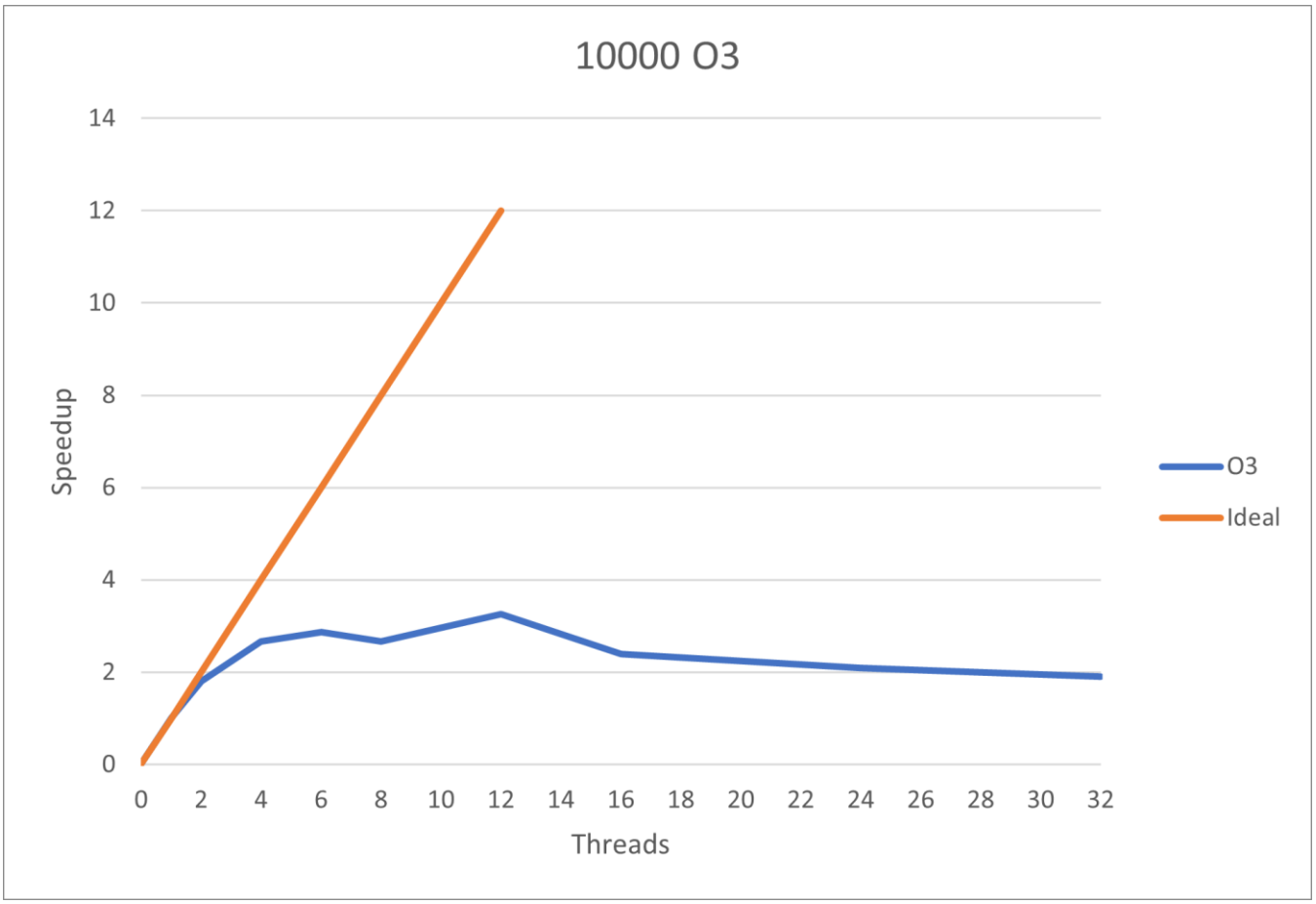
Misure O2-10000

Type	Time	Speedup
Sequential	*0,016967*	*/*
Threads 1	*0,01716*	*0,988711*
Threads 2	*0,009261*	*1,832133*
Threads 4	*0,006216*	*2,729422*
Threads 6	*0,005514*	*3,076866*
Threads 8	*0,005952*	*2,850682*
Threads 12	*0,005437*	*3,120531*
Threads 16	*0,005694*	*2,979614*
Threads 24	*0,008225*	*2,062746*
Threads 32	*0,00901*	*1,883075*



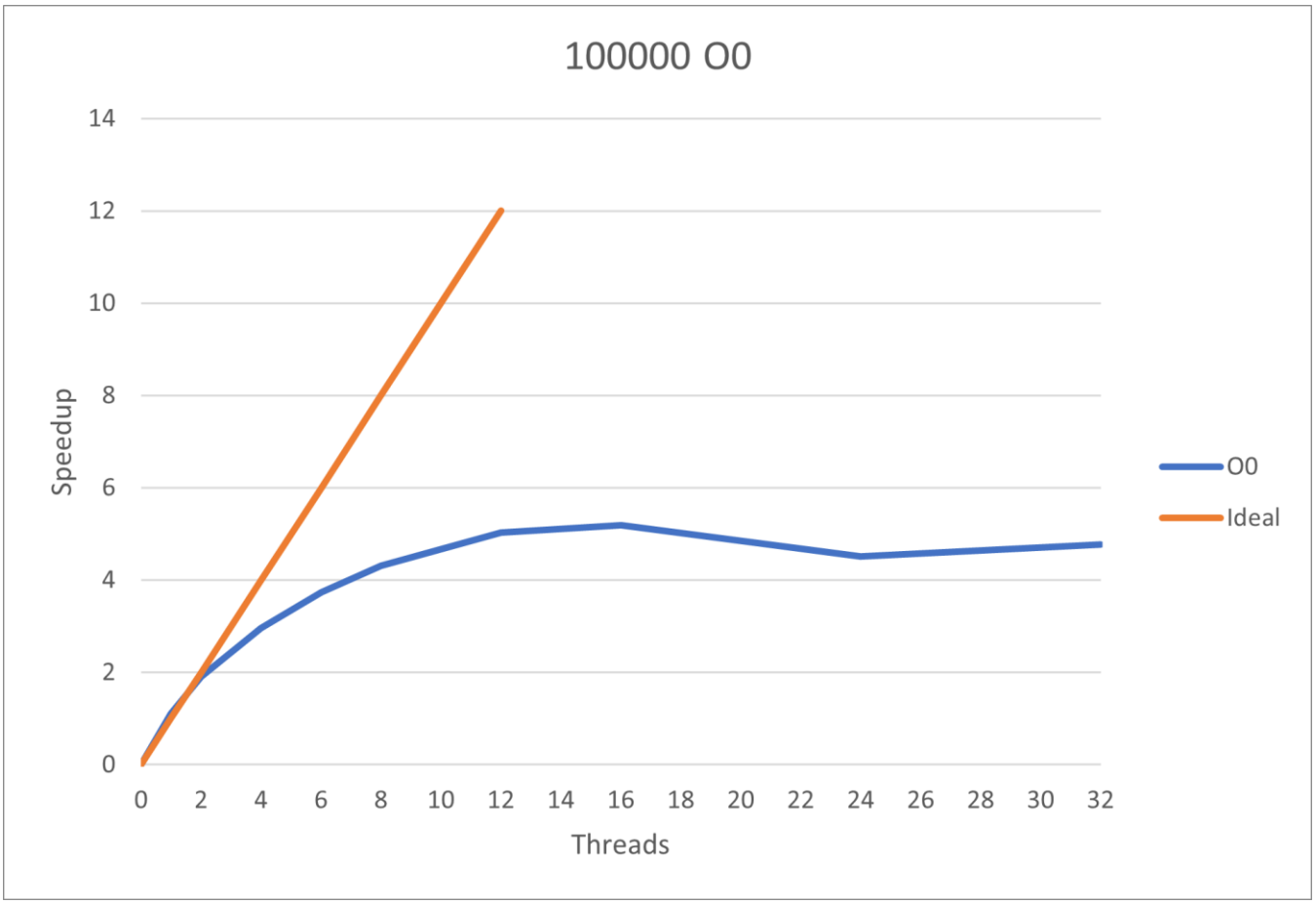
Misure O3-10000

Type	Time	Speedup
Sequential	*0,016932*	*/*
Threads 1	*0,016705*	*1,013635*
Threads 2	*0,00941*	*1,79933*
Threads 4	*0,006346*	*2,668258*
Threads 6	*0,005918*	*2,860975*
Threads 8	*0,006344*	*2,669217*
Threads 12	*0,005203*	*3,254192*
Threads 16	*0,007088*	*2,388726*
Threads 24	*0,008128*	*2,083252*
Threads 32	*0,008908*	*1,900827*



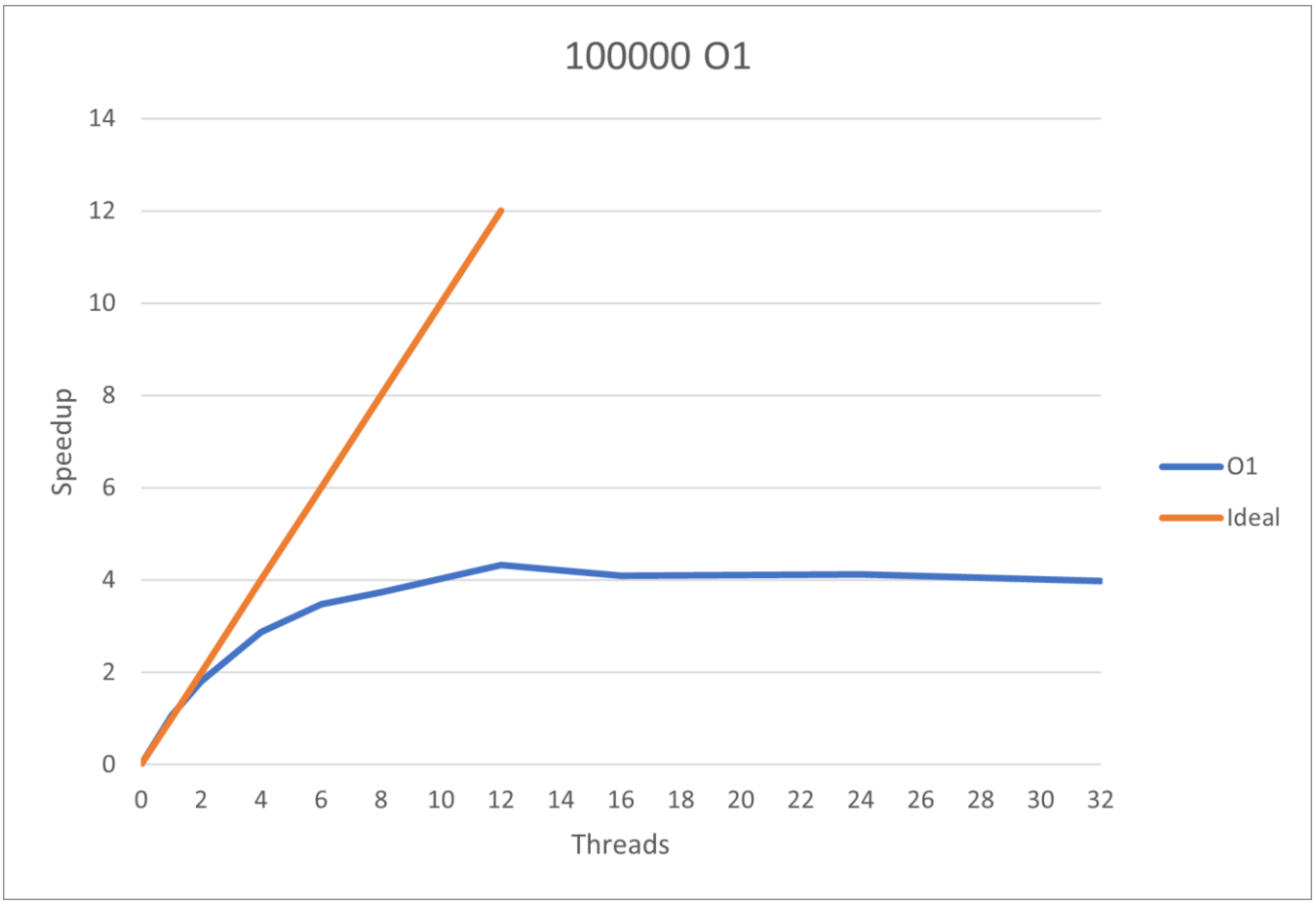
Misure O0-100000

Type	Time	Speedup
Sequential	*0,407263*	*/*
Threads 1	*0,368938*	*1,103879*
Threads 2	*0,212935*	*1,912613*
Threads 4	*0,137853*	*2,95433*
Threads 6	*0,108846*	*3,741646*
Threads 8	*0,094489*	*4,310158*
Threads 12	*0,080926*	*5,032554*
Threads 16	*0,078457*	*5,190888*
Threads 24	*0,090371*	*4,50658*
Threads 32	*0,085386*	*4,769665*



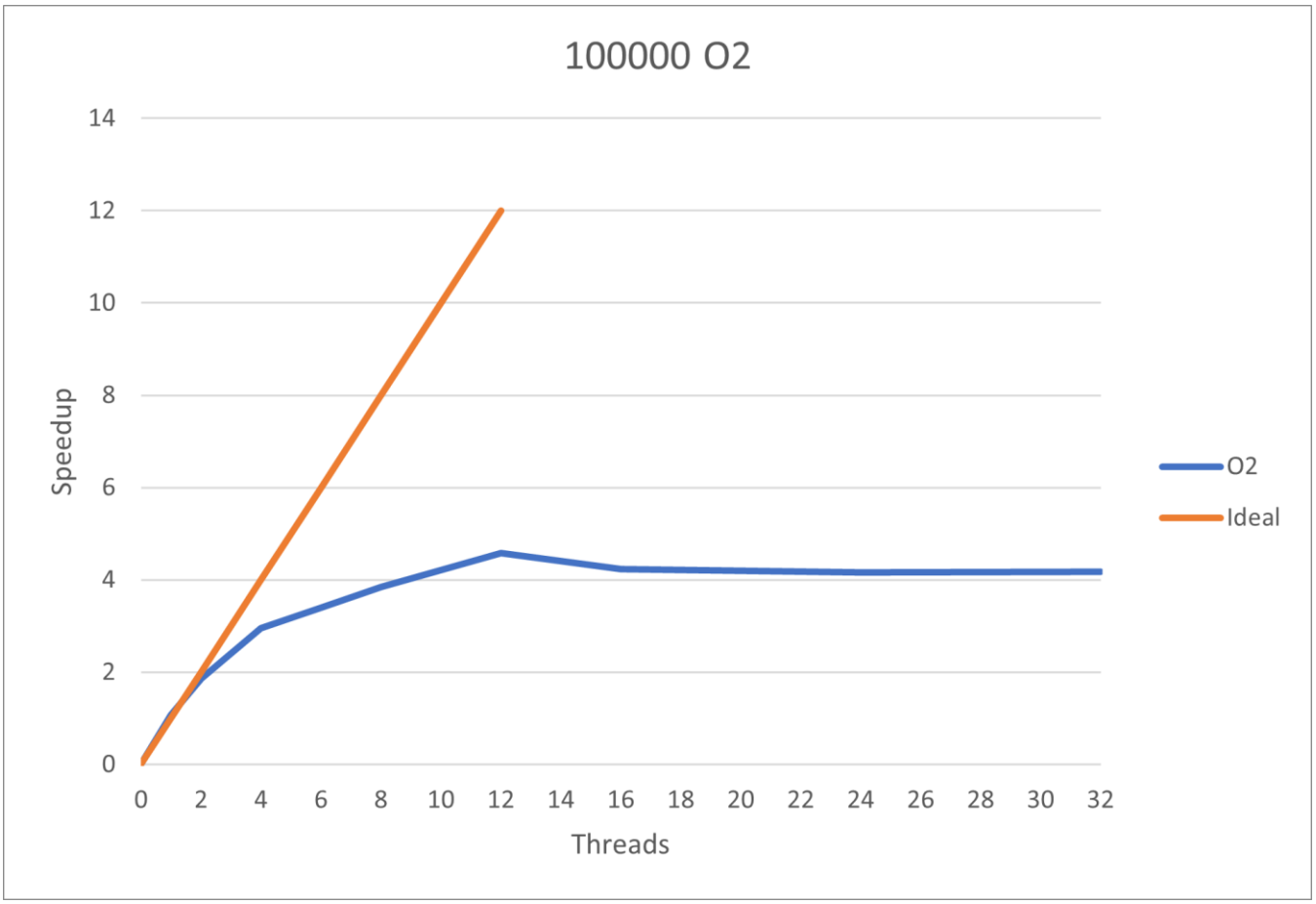
Misure O1-100000

Type	Time	Speedup
Sequential	*0,344207*	*/*
Threads 1	*0,324736*	*1,059961*
Threads 2	*0,190065*	*1,810996*
Threads 4	*0,120026*	*2,867776*
Threads 6	*0,099041*	*3,475413*
Threads 8	*0,092146*	*3,735463*
Threads 12	*0,07969*	*4,319346*
Threads 16	*0,083955*	*4,099914*
Threads 24	*0,083574*	*4,118571*
Threads 32	*0,086438*	*3,982131*



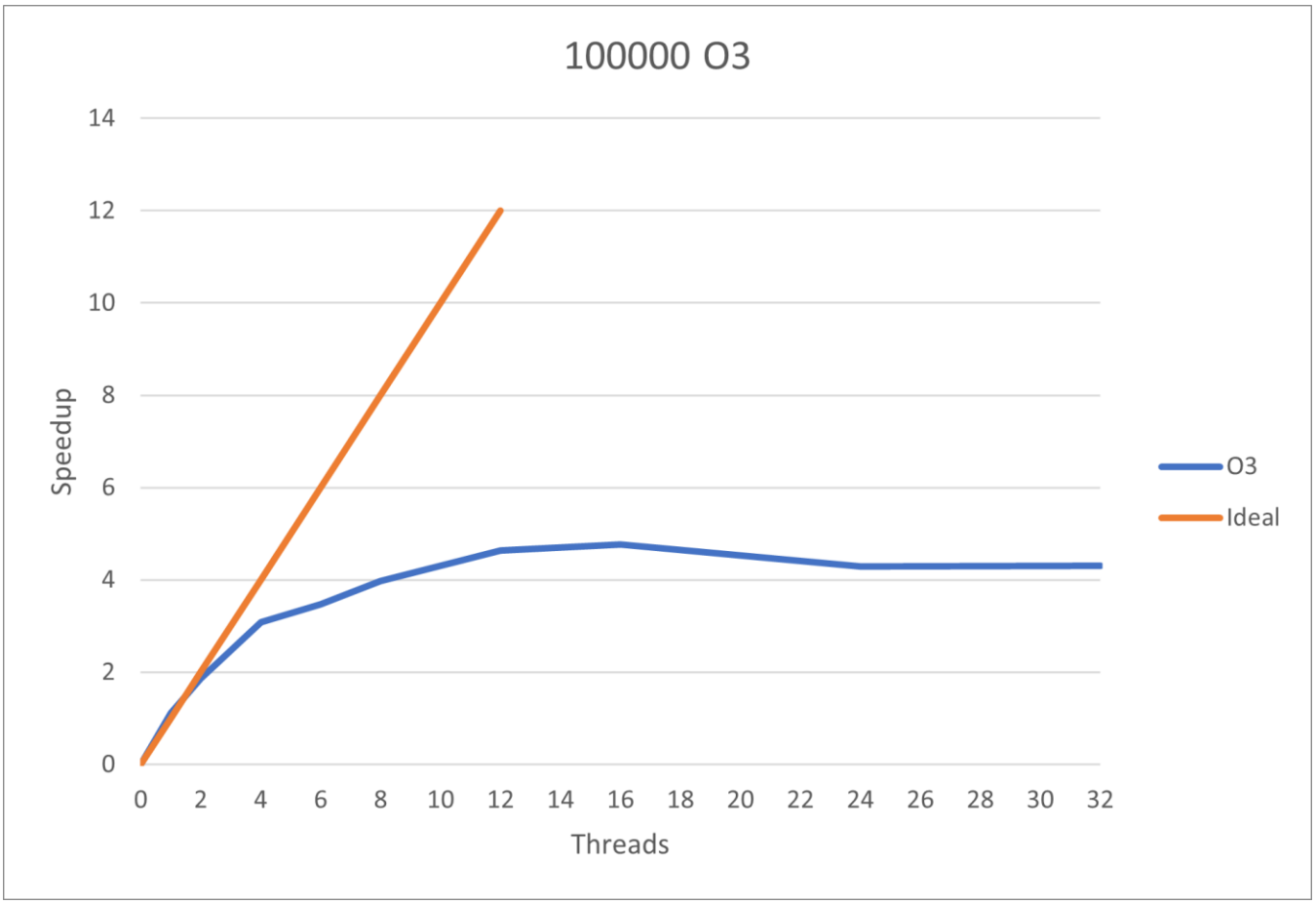
Misure O2-100000

Type	Time	Speedup
Sequential	*0,352154*	*/*
Threads 1	*0,327213*	*1,076224*
Threads 2	*0,189336*	*1,859939*
Threads 4	*0,119539*	*2,94594*
Threads 6	*0,103571*	*3,400128*
Threads 8	*0,09161*	*3,844045*
Threads 12	*0,07702*	*4,57225*
Threads 16	*0,083088*	*4,238304*
Threads 24	*0,084665*	*4,159377*
Threads 32	*0,084286*	*4,178066*



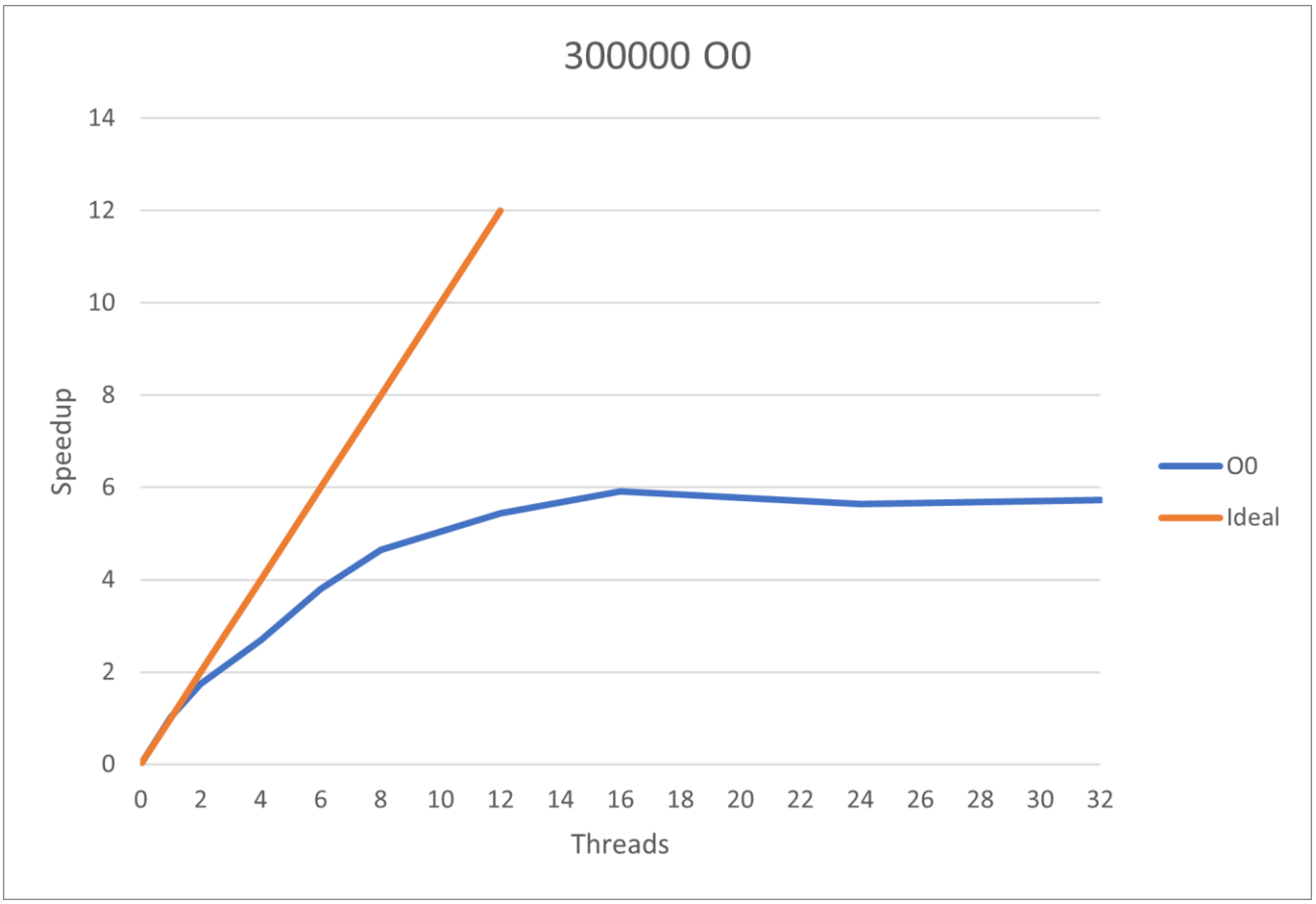
Misure O3-100000

Type	Time	Speedup
Sequential	*0,358881*	*/*
Threads 1	*0,323981*	*1,107721*
Threads 2	*0,192297*	*1,866282*
Threads 4	*0,116632*	*3,077028*
Threads 6	*0,103615*	*3,463597*
Threads 8	*0,090298*	*3,974414*
Threads 12	*0,077517*	*4,629707*
Threads 16	*0,075182*	*4,773527*
Threads 24	*0,08364*	*4,290784*
Threads 32	*0,083442*	*4,300985*



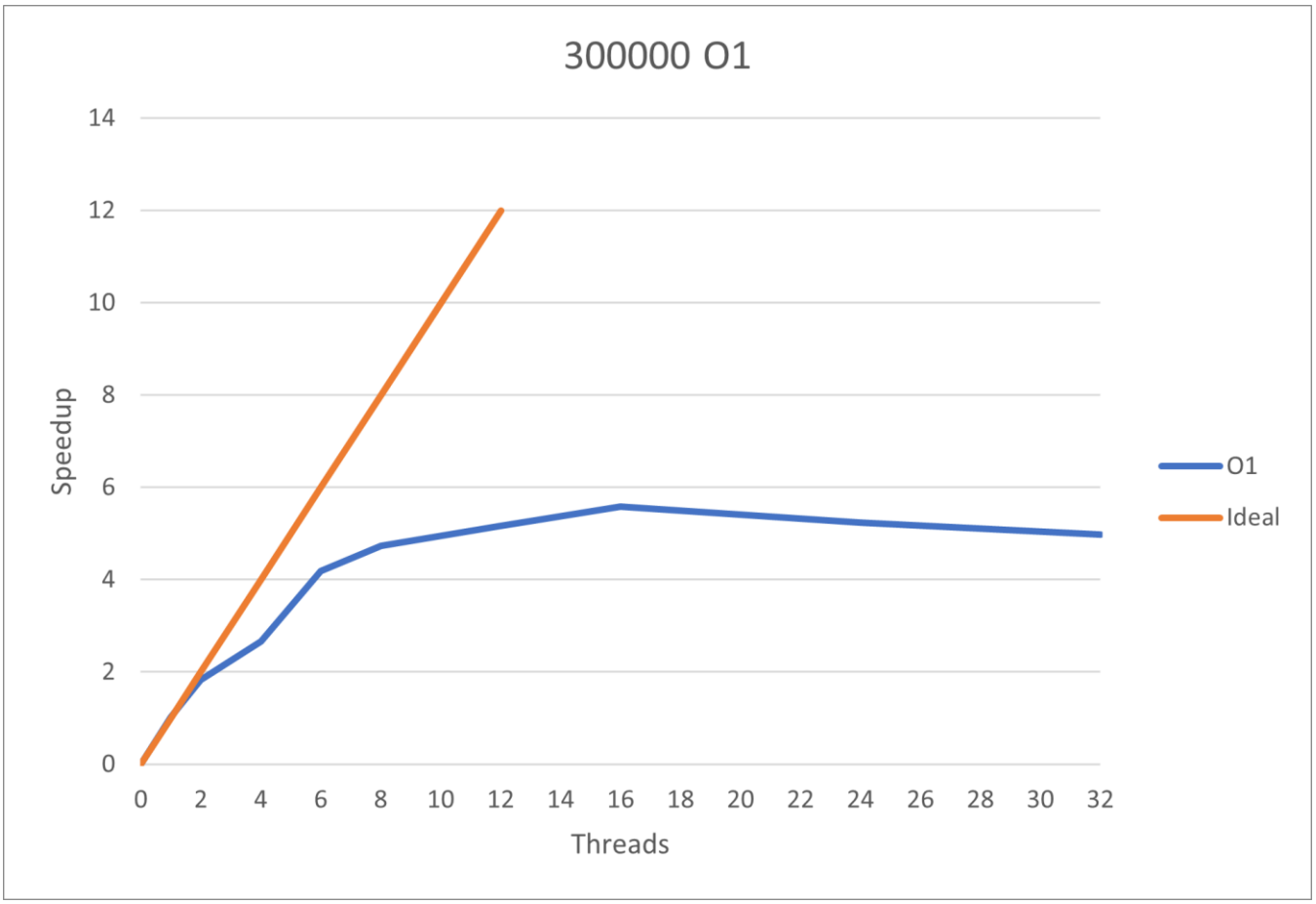
Misure O0-300000

Type	Time	Speedup
Sequential	*2,067599*	*/*
Threads 1	*2,008214*	*1,029571*
Threads 2	*1,184562*	*1,745454*
Threads 4	*0,766747*	*2,696586*
Threads 6	*0,544569*	*3,796759*
Threads 8	*0,444737*	*4,64904*
Threads 12	*0,379899*	*5,442495*
Threads 16	*0,349426*	*5,91713*
Threads 24	*0,366401*	*5,642993*
Threads 32	*0,36097*	*5,727893*



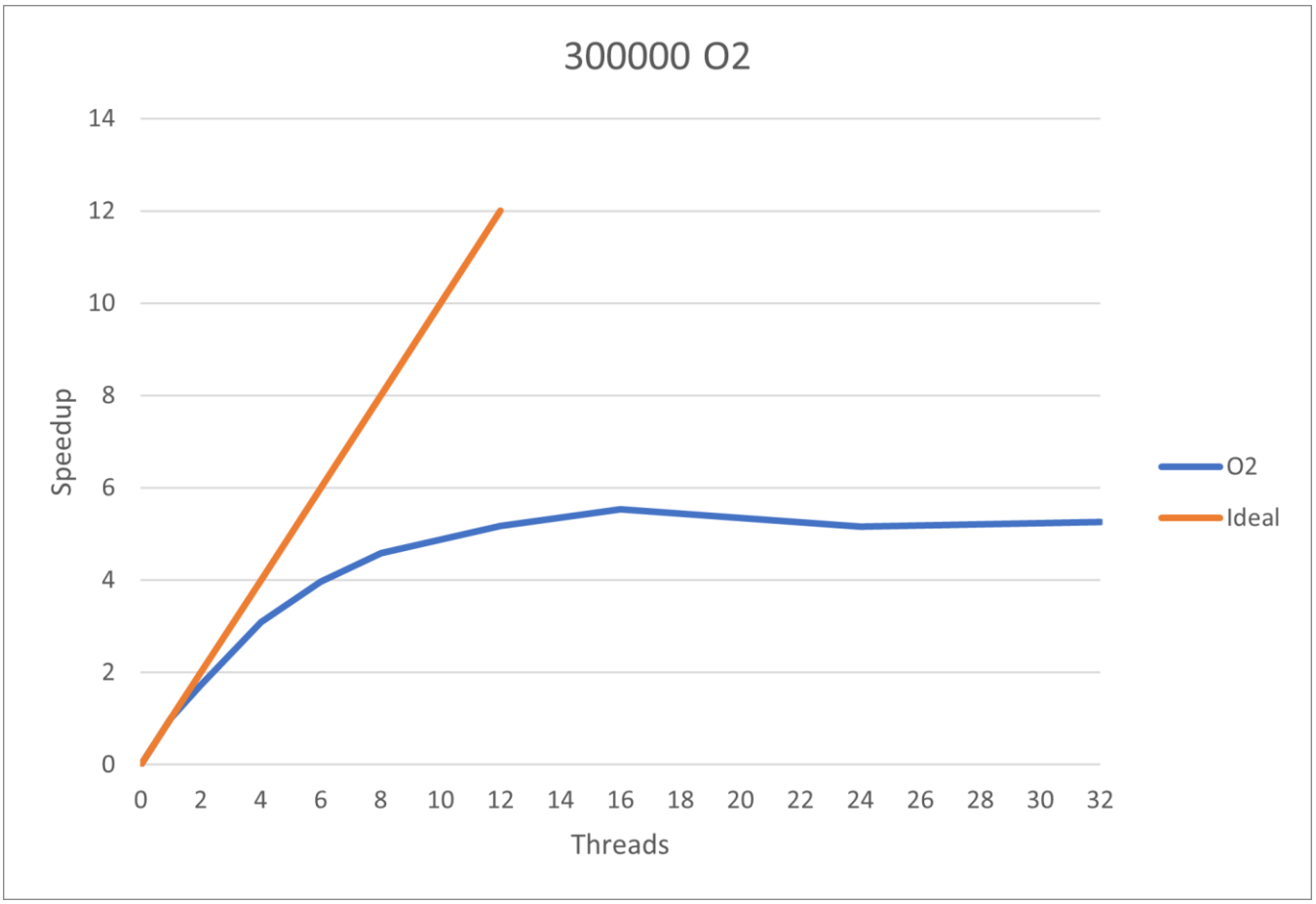
Misure O1-300000

Type	Time	Speedup
Sequential	*1,875561*	*/*
Threads 1	*1,840121*	*1,01926*
Threads 2	*1,02925*	*1,82226*
Threads 4	*0,706102*	*2,656221*
Threads 6	*0,447534*	*4,190882*
Threads 8	*0,395975*	*4,73657*
Threads 12	*0,363229*	*5,163573*
Threads 16	*0,335942*	*5,582989*
Threads 24	*0,357818*	*5,241666*
Threads 32	*0,376537*	*4,981085*



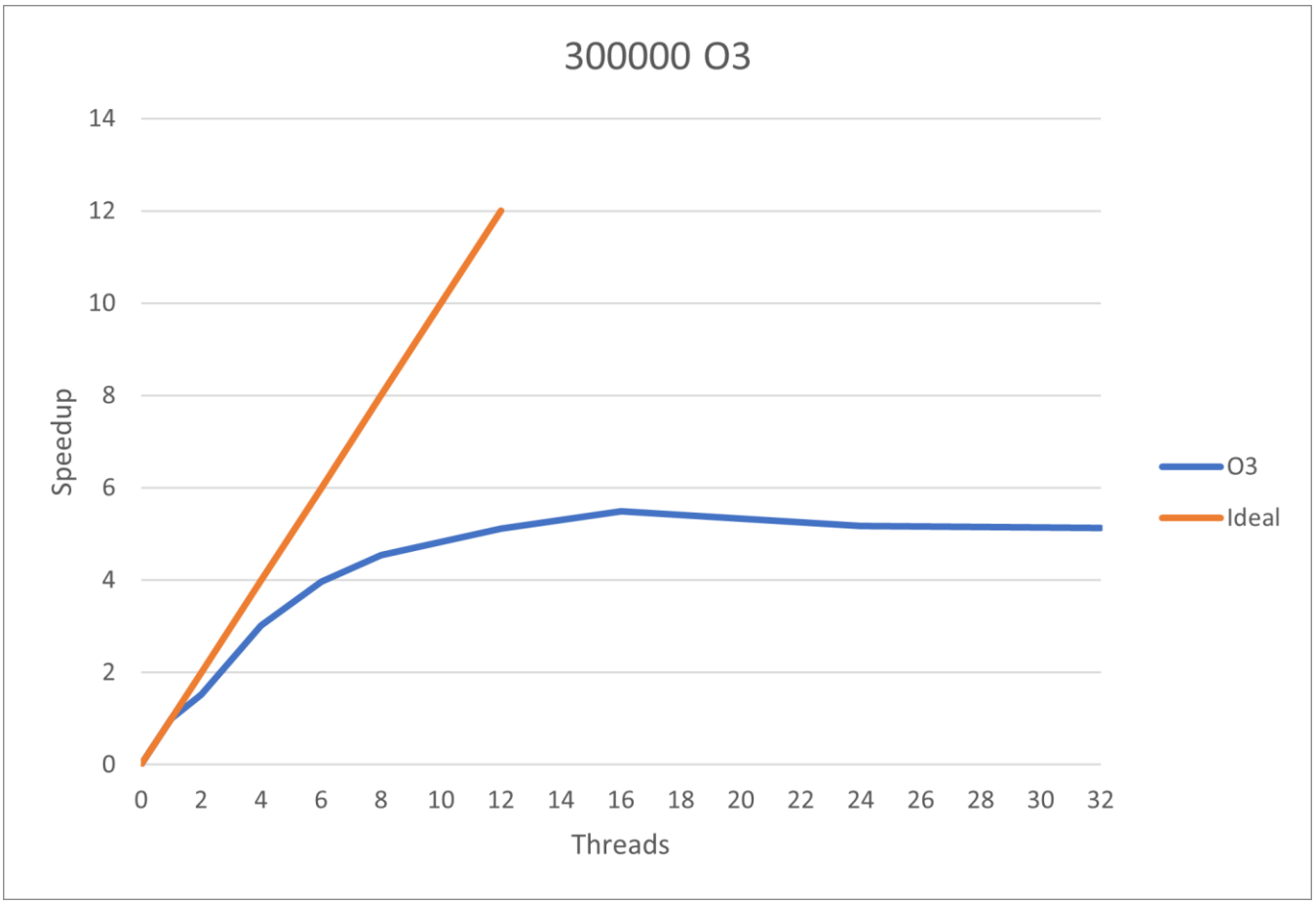
Misure O2-300000

Type	Time	Speedup
Sequential	*1,860407*	*/*
Threads 1	*1,857083*	*1,00179*
Threads 2	*1,082579*	*1,718496*
Threads 4	*0,602999*	*3,085255*
Threads 6	*0,468686*	*3,969414*
Threads 8	*0,405427*	*4,588756*
Threads 12	*0,359294*	*5,177955*
Threads 16	*0,336475*	*5,529105*
Threads 24	*0,360366*	*5,162546*
Threads 32	*0,35336*	*5,264907*



Misure O3-300000

Type	Time	Speedup
Sequential	*1,829616*	*/*
Threads 1	*1,815647*	*1,007694*
Threads 2	*1,208207*	*1,514323*
Threads 4	*0,606482*	*3,016768*
Threads 6	*0,460693*	*3,971448*
Threads 8	*0,402322*	*4,547639*
Threads 12	*0,358083*	*5,109477*
Threads 16	*0,332823*	*5,497269*
Threads 24	*0,354015*	*5,168194*
Threads 32	*0,356258*	*5,135644*



Conclusioni e considerazioni

L'algoritmo implementato risulta molto efficiente su grafi con molti vertici, con un andamento crescente fino a 16 threads, oltre questa soglia lo speedup tende a rimanere costante.

Nel caso in cui ci sono pochi vertici (come nel caso di 1000 vertici), il sequenziale risulta molto più efficace dovuto principalmente al fatto che il tempo di esecuzione è davvero basso. Di conseguenza, ogni tecnica utilizzata per la parallelizzazione (array e altri elementi) risulta costosa in termini di tempo per l'istanziamento e l'allocazione.

L'ottimizzazione che presenta uno speedup maggione è O0, questo è dovuto principalmente all'abbassamento del tempo di esecuzione dell'algoritmo sequenziale.

Infatti, prendendo in considerazione le misurazioni su 100'000 vertici, abbiamo:

type	O0	O1	O2	O3
Sequential	*0,407263*	*0,344207*	*0,352154*	*0,358881*
Threads 12	*0,080926*	*0,07969*	*0,07702*	*0,077517*

Da questa tabella si evince che entrambi gli algoritmi risultano migliorati nell'ottimizzazione, ma quello sequenziale subisce un'ottimizzazione maggiore.

Come eseguire i test

1) Crea una cartella chiama “***build***” e lancia il comando “***cmake***”:

```
mkdir  build
cd     build
cmake ..
```

2) Generare gli eseguibili con il comando “***make***”:

```
make
```

3) Per generare le varie misure (le misure descritte in questo documento sono già presenti all’interno della cartella “*measure_bfs*”) bisogna utilizzare il seguente comando

```
make generate_output
```

4) Per poter visualizzare i dati relativi alle misurazioni, aprire il file excel “BFS_measure_OMP”, dirigersi in “Dati” e cliccare su “Aggiorna tutti”. In questo modo automaticamente verrano presi i dati e inseriti in excel per poterli vedere sia in formato testuale, sia sottoforma di grafici.