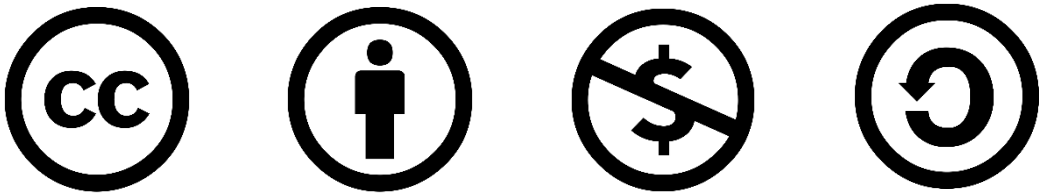


Report Common Assignment 1

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 message passing: MPI.

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

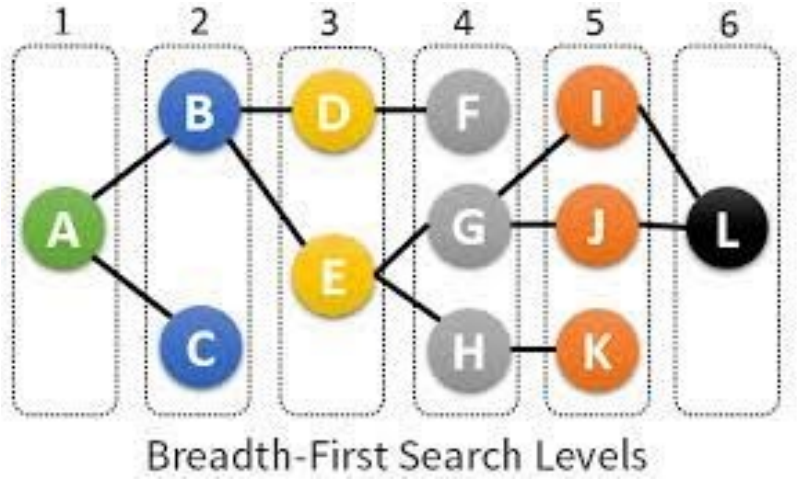
Riguardo le misure

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, 5000, 10000, 20000.

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.



Algoritmo 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* q, struct queue* bfs_queue) {
  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);

    for(int i=0; i<graph->numVertices; i++){
      if(graph->adjMatrix[currentVertex][i] == 1){
        if (graph->visited[i] == 0) {
          graph->visited[i] = 1;
          enqueue(q, i);
          enqueue(bfs_queue, i);
        }
      }
    }
  }
}
```

Algoritmi di parallelizzazione

Esistono diversi algoritmi di parallelizzazione della BFS su memoria distribuita. Gli algoritmi più diffusi sono ***1D*** e ***2D Partitioning.*** Questi algoritmi lavorano su grafi rappresentati tramite matrice di adiacenza. Viene divisa la matrice di adiacenza in modo tale che ogni processore può effettuare le operazioni su un sottoinsieme di vertici.

Partizionamento 1D

Nel partizionamento 1D, le righe della matrice vengono assegnate in modo casuale ai processori nel sistema. Pertanto, ogni riga contiene l'elenco completo dei vicini del vertice corrispondente. La DBFS (Distributed BFS) attraversa tutti i vertici nel grafo ordinati in base alla loro distanza da un vertice sorgente v_s. Per questo motivo, l'algoritmo è sincronizzato per livelli, ovvero i nodi devono essere sincronizzati dopo aver attraversato i vertici a una certa distanza rispetto a *v*, per garantire che i vertici più vicini al vertice di origine vengano visitati per primi. DBFS funziona in questo modo: ogni processo p_i esegue lo stesso codice sulla sua partizione v_i e riceve come parametro il vertice sorgente v_s. Dato un processo p_i, vengono mantenuti tre sottoinsiemi di vertici per nodo:

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	1	1	0
2	0	0	1	0	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	1	0
7	0	0	1	0	0	1	0	1	0	1	1	0
8	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1
10	1	0	0	0	0	0	1	0	0	0	1	0
11	1	0	0	0	0	1	1	0	1	1	0	0
12	0	0	0	1	1	0	0	0	1	0	0	0

(a) 1D Partitioning

1) Il sottoinsieme dei vertici visitati $V_{visited} \in V_i$,
2) I vertici da visitare durante l'iterazione corrente $V_{current} \in V_i$ (cioè i vertici posti alla stessa distanza minima dal vertice sorgente nel nodo locale),
3) I vertici da visitare nell'iterazione successiva $V_{next} \in V_i$.

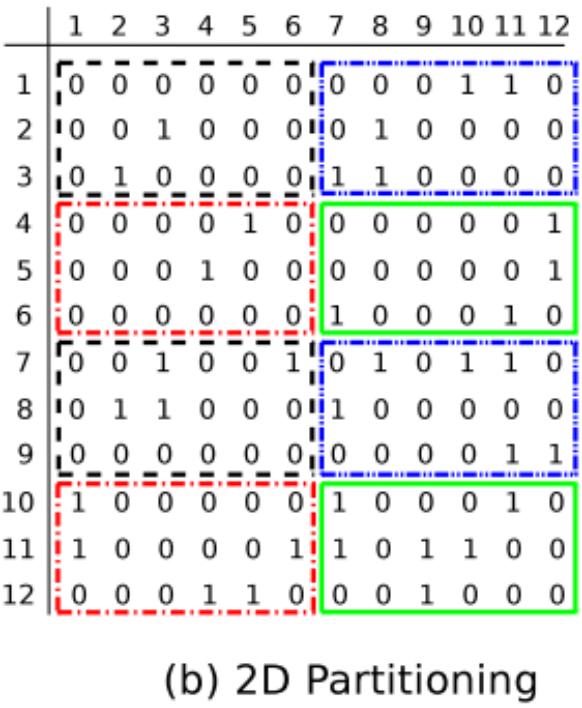
Il ciclo esterno viene eseguito fino a quando non ci sono più vertici da sfruttare in nessuno dei nodi nel sistema. Per ogni vertice $v \in V_{current}$,, si trova il suo insieme di vertici vicini N_v. Per ogni vertice $u \in N_v$, se u è assegnato al nodo locale e non è stato precedentemente visitato, viene aggiunto a V_{next}, altrimenti l'identificatore di vertice viene inviato a N_x, essendo $u \in N_x$.

Algorithm 1. Distributed BFS (DBFS) with 1D partitioning

```
1:  procedure DBFS(v_s)                                   ▷ Executed in each node N_i
2:      V_visited = ∅; V_next = ∅;                        ▷ Subsets of vertices are initiallized
3:      if v_s ∈ V_i then V_current = {v_s};
4:      else V_current = ∅;
5:      end if
6:      while at least 1 node has V_current ≠ ∅ do
7:          for all v ∈ V_current do
8:              N_v = neighbors(v) \ (V_visited ∪ V_current);   ▷ Neighbors of v not found yet
9:              for all u ∈ N_v ∩ B do                    ▷ Boundaries are sent to their nodes
10:                 send u to N_x such that u ∈ N_x;
11:             end for
12:             V_visited = V_visited ∪ {v}; V_next = V_next ∪ (N_v ∩ V_i);
13:         end for
14:         for all u ∈ B sent by other nodes N_x do
15:             receive u from N_x;                       ▷ Receive boundaries assigned to the current node
16:             V_next = V_next ∪ {u} \ (V_visited ∪ V_current);
17:         end for
18:         wait until all nodes have processed their corresponding V_current;
19:         V_current = V_next; V_next = ∅;
20:     end while
21: end procedure
```

Partizionamento 2D

In DBFS con partizionamento 2D, l'idea è di trarre vantaggio dal fatto che i nodi sono organizzati in mesh di processori $R \times C = p$ nell'architettura del sistema che stanno utilizzando, cioè in una griglia bidimensionale tale che ogni nodo è connesso ai suoi quattro vicini immediati. La matrice di adiacenza è suddivisa in $R \times C$ blocchi di righe e C blocchi di colonne $\{CB_1, \dots, CB_C\}$, in modo tale che a ciascun nodo siano assegnati quei valori nella matrice di adiacenza corrispondenti all'intersezione tra C blocchi di righe e un blocco di colonne. Pertanto, l'elenco delle adiacenze di un dato vertice è diviso in R nodi e l'insieme dei vertici nel grafo è assegnato a gruppi C di R nodi. Per far esplorare un vertice v da un nodo in CB_i, viene inviato un messaggio agli altri $R - 1$ nodi in CB_j per ottenere l'elenco completo delle adiacenze di quel vertice e ciascuno di questi nodi comunica con uno degli altri $C - 1$ nodi nella stessa riga della mesh, se i vertici adiacenti di v appartengono a un altro CB_j. Grazie a ciò, il numero di nodi a cui un certo nodo invia dati può essere ridotto drasticamente, poiché ogni nodo comunica al massimo solo con $R + C - 2$ anziché con p nodi come nel caso del partizionamento 1D.



	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	1	1	0
2	0	0	1	0	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	1	0
7	0	0	1	0	0	1	0	1	0	1	1	0
8	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1
10	1	0	0	0	0	0	1	0	0	0	1	0
11	1	0	0	0	0	1	1	0	1	1	0	0
12	0	0	0	1	1	0	0	0	1	0	0	0

(b) 2D Partitioning

Implementazione dell'algoritmo 1D Partitioning

Punti di forza

Il punto di forza più evidente è la semplicità nella realizzazione dato che i passi da seguire sono molto simili all'algoritmo sequenziale con alcune differenze dovute alla memoria distribuita.

Punti di debolezza

Il punto di debolezza principale sta nella matrice di adiacenza, infatti, se il grafo è poco denso, ci sarà un grande tasso di spazio sprecato ma soprattutto messaggi da inviare molto grandi con poche informazioni importanti. Inoltre, ogni sottomatrice ha le informazioni sui soli vertici di appartenenza, di conseguenza, se un processore ha n vertici e ognuno ha un vicino che appartiene ad un altro processore, esso verrà considerato n volte per poi essere ignorato almeno n-1 volte dal processore di appartenenza del nodo.

Implementazione e dettagli specifici

Il primo passo è stato quello di creare una sottomatrice di adiacenza per ogni processo e, tramite un MPI_scatter, suddividere la matrice di adiacenza originaria.

```
//Istanzio la sottomatrice di adiacenza del grafo
int *subAdjMatrix = (int *) malloc(sizeof(int) * avg_num_of_vertices_per_proc * n_vertices);

//Suddivido la matrice di adiacenza per ogni processo
MPI_Scatter(adjMatrix, avg_num_of_vertices_per_proc * n_vertices, MPI_INT, subAdjMatrix, avg_num_of_vertices_per_proc * n_vertices, MPI_INT, 0, MPI_COMM_WORLD);
```

Il secondo passo è quello di settare ogni livello per processore pari a 0, tranne lo START_NODE, il quale starà nel livello 1.

```
//Per ogni processo, controllo se sono il proprietario dello START_NODE
//Se lo sono, allora lo setto come visitato e con valore pari a 1
for(int i = 0; i < avg_num_of_vertices_per_proc; i++){
    if(first_vertex + i == START_NODE){
        level[i] = 1;
    }
    else
        level[i] = 0;
}
```

Successivamente è necessario creare la coda F e N, rispettivamente l'insieme dei vertici da esplorare (frontiera) e dei vicini trovati (neighbours).

```
//Creo le due code per ogni processo
struct queue* F = createQueue(n_vertices);
struct queue* N = createQueue(n_vertices);

//Creo un array per poter immagazzinare i vari vicini visitati per ogni processo
struct queue** map = (struct queue**)malloc(sizeof(struct queue*) * size);
for(int i = 0; i < size; i++){
  map[i] = createQueue(n_vertices);
}
```

Dopo aver avviato la preparazione di tutti gli elementi necessari al funzionamento, si può avviare il reale algoritmo. È presente un ciclo for dove ogni iterazione rappresenta il livello attuale.

Per ogni livello, quindi, ci sono vari passi da seguire. Il primo consiste nell'inserimento dei vertici di appartenenza del processo in coda. Vengono inseriti solo i vertici che appartengono al livello di esplorazione attuale.

```
for(int l = 1; l < n_vertices; l++){
    //Sincronizzo i processori
    MPI_Barrier(MPI_COMM_WORLD);

    //Resetto le code
    reset_queue(F);
    reset_queue(N);

    //Metto in coda i vertici di proprietà del processore attuale
    for(int i = 0; i < num_of_vertices_per_proc; i++){
      if(level[i] == l){
        enqueue(F, first_vertex + i);
      }
    }
```

Successivamente, è necessario controllare se nella frontiera attuale sono presenti dei vertici. Se esiste almeno un vertice da esplorare in tutti i processori, allora l'iterazione prosegue, altrimenti la BFS viene fermata.

```
//Controllo se il processore ha almeno un vertice da esplorare
if(isEmpty(F))
    frontier_has_vertices = 0;
else
    frontier_has_vertices = 1;

//Raccolgo le informazioni da tutti i processori
MPI_Allreduce(&frontier_has_vertices, &continue_search, 1, MPI_INT, MPI_LOR, MPI_COMM_WORLD);

//Termina il ciclo se non ci sono vertici da espolare in tutti i processori
if(!continue_search){
    break;
}
```

Per ogni vertice nella frontiera vengono esplorati i vicini e inseriti nella coda N. Questo avviene se il vertice vicino non è stato esplorato in un altro livello.

```
//Trova i vicini di ogni vertice del livello attuale
while(!isEmpty(F)){
  currentVertex = dequeue(F);
  rel_vertex = (currentVertex % avg_num_of_vertices_per_proc);
  for(int i = 0; i < n_vertices; i++){
    if(subAdjMatrix[rel_vertex * n_vertices + i] != 0){
      if(i >= first_vertex && i < first_vertex + avg_num_of_vertices_per_proc){
        if(level[i%avg_num_of_vertices_per_proc] == 0){
          enqueue_if_not_exist(N, i);
        }
      }
      else{
        enqueue_if_not_exist(N, i);
      }
    }
  }
}
```

Una volta ottenuti tutti i vicini, vengono smistati in delle code. Ogni coda rappresenta la coda dei vertici di appartenenza di un determinato processore.

```
//Resetto map
for(int i = 0; i < size; i++){
  reset_queue(map[i]);
}

//Inserisco i vertici trovati nell'indice del processore proprietario del vertice
while(!isEmpty(N)){
  currentVertex = dequeue(N);
  enqueue(map[currentVertex / avg_num_of_vertices_per_proc], currentVertex);
}
```

Calcolo il displacement e la dimensione del buffer di trasmissione.

```
//Calcolo il displacement e le dimensioni
for(int i = 0; i < size; i++){
    send_disps[i] = send_buf_size;
    send_buf_size += n_element_queue(map[i]);
    send_counts[i] = n_element_queue(map[i]);
}
```

Creo il buffer di trasmissione e associo tutti i valori da trasmettere. Trasmetto a tutti i processori la dimensione del buffer.

```
//Creo il buffer per l'invio
int k = 0;
send_buf = (int *)malloc(sizeof(int) * send_buf_size);
for(int i = 0; i < size; i++){
  for(int j = 0; j < send_counts[i] ; j++){
      send_buf[k] = dequeue(map[i]);
      k++;
  }
}

//Invio ad ogni processore il numero di vertici complessivi trovati
MPI_Alltoall(send_counts, 1, MPI_INT, recv_counts, 1, MPI_INT, MPI_COMM_WORLD);
```

Calcolo la dimensione e il displacement del buffer in ricezione e poi invio a tutti i processori il proprio buffer contenente tutti i vertici di proprietà del singolo processore. Successivamente viene aggiornato level.

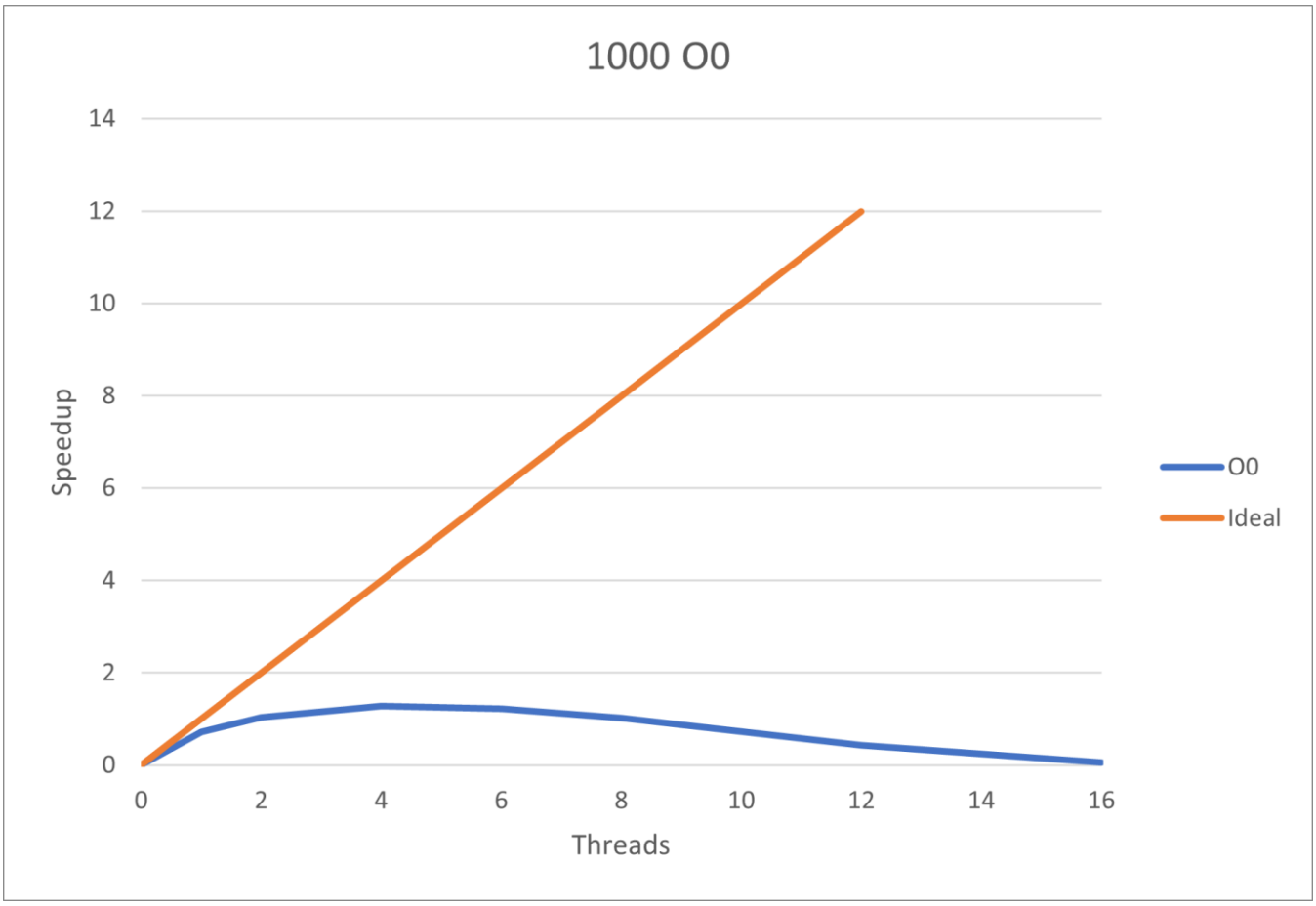
```
//Calcolo la dimensione e il displacement dell'array ricevuto
for(int i = 0; i < size; i++){
    recv_disps[i] = recv_buf_size;
    recv_buf_size += recv_counts[i];
}

//Invio ad ogni processore tutti i vertici visitati che sono di loro proprietà
recv_buf = (int *)malloc(sizeof(int) * recv_buf_size);
MPI_Alltoallv(send_buf, send_counts, send_disps, MPI_INT,
        recv_buf, recv_counts, recv_disps, MPI_INT, MPI_COMM_WORLD);

//Smisto i vertici trovati ad ogni processore
for(int i = 0; i < recv_buf_size; i++){
    rel_vertex = recv_buf[i] % avg_num_of_vertices_per_proc;
    if(level[rel_vertex] == 0)
        level[rel_vertex] = l + 1;
}
```

Viene iterato il tutto finchè non ci sono più vertici da esplorare.

Analisi delle misure

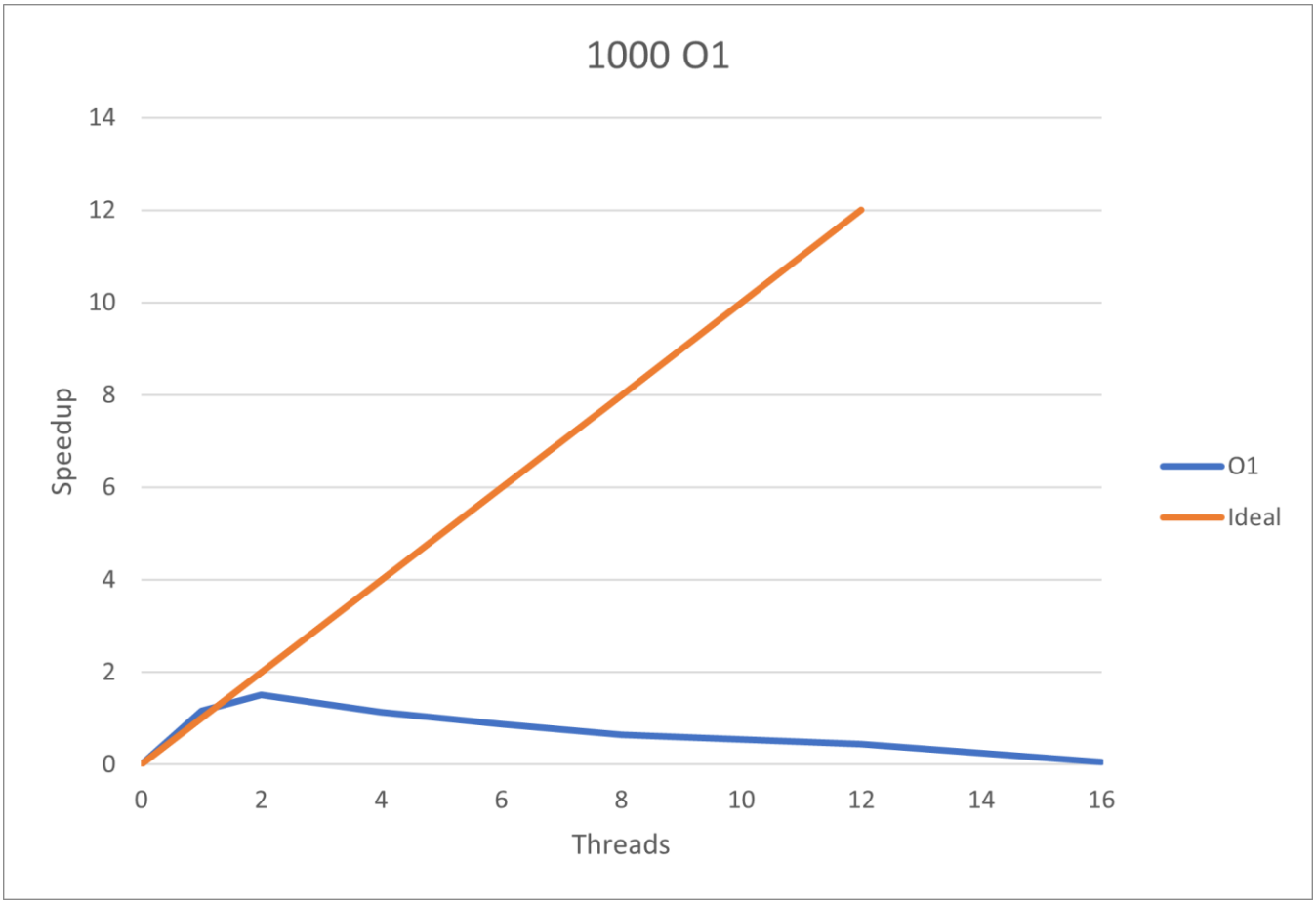
Misure O0-1000

Type	Time	Speedup
Sequential	*0,001682*	*/*
Threads 1	*0,002359*	*0,713059*
Threads 2	*0,001619*	*1,038831*
Threads 4	*0,00131*	*1,283804*
Threads 6	*0,001376*	*1,222277*
Threads 8	*0,00166*	*1,013349*
Threads 12	*0,003938*	*0,427148*
Threads 16	*0,030468*	*0,055212*



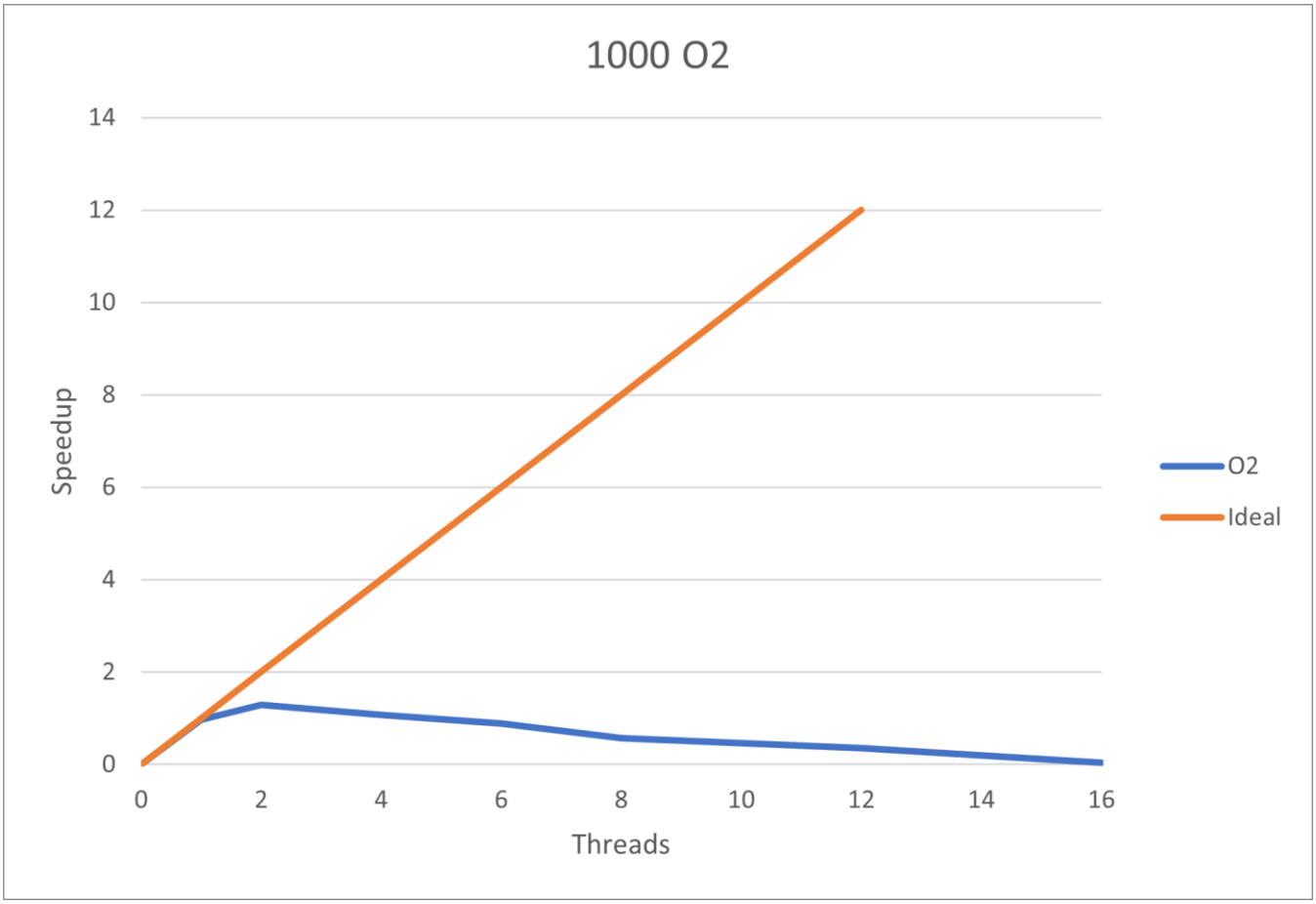
Misure O1-1000

Type	Time	Speedup
Sequential	*0,000875*	*/*
Threads 1	*0,000754*	*1,160724*
Threads 2	*0,000579*	*1,512323*
Threads 4	*0,000767*	*1,140095*
Threads 6	*0,000998*	*0,877123*
Threads 8	*0,001355*	*0,64569*
Threads 12	*0,001952*	*0,448267*
Threads 16	*0,017033*	*0,051371*



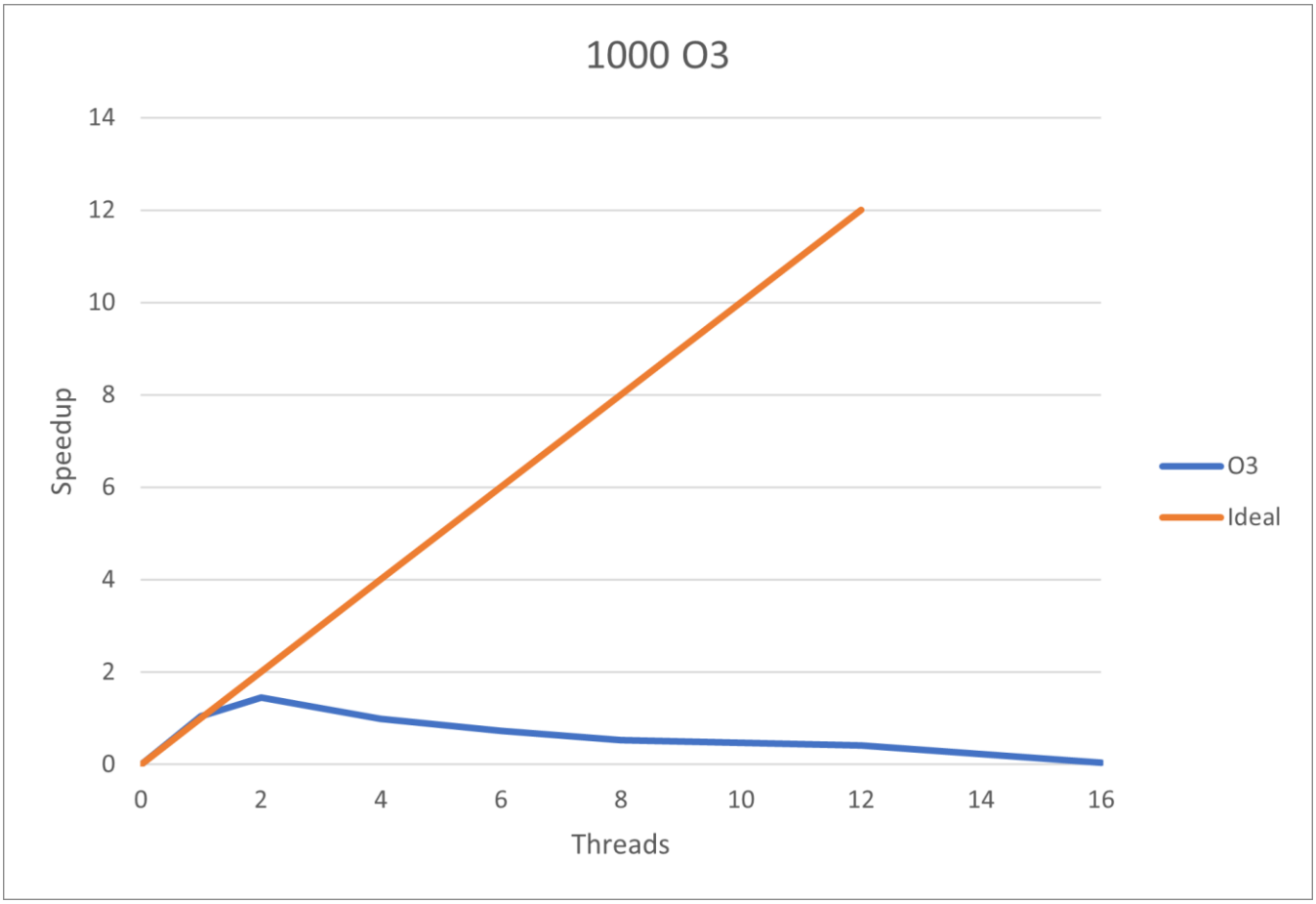
Misure O2-1000

Type	Time	Speedup
Sequential	*0,000758*	*/*
Threads 1	*0,000784*	*0,967193*
Threads 2	*0,000591*	*1,283186*
Threads 4	*0,000707*	*1,072655*
Threads 6	*0,000859*	*0,882416*
Threads 8	*0,001321*	*0,5739*
Threads 12	*0,002141*	*0,354158*
Threads 16	*0,017955*	*0,042231*



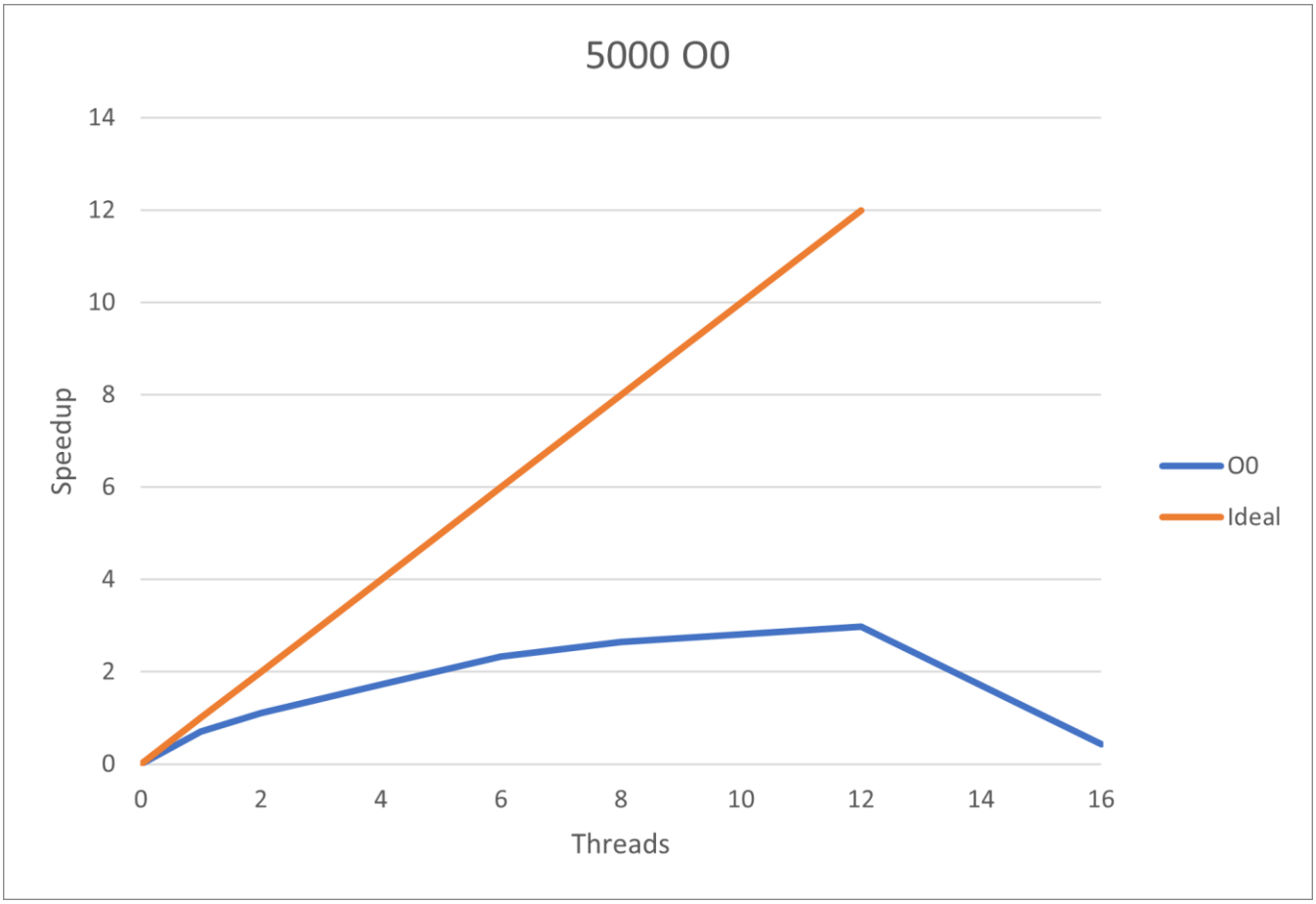
Misure O3-1000

Type	Time	Speedup
Sequential	*0,000745*	*/*
Threads 1	*0,000715*	*1,042317*
Threads 2	*0,000517*	*1,440473*
Threads 4	*0,000754*	*0,988507*
Threads 6	*0,001019*	*0,731124*
Threads 8	*0,001423*	*0,523437*
Threads 12	*0,001826*	*0,408011*
Threads 16	*0,018297*	*0,040709*



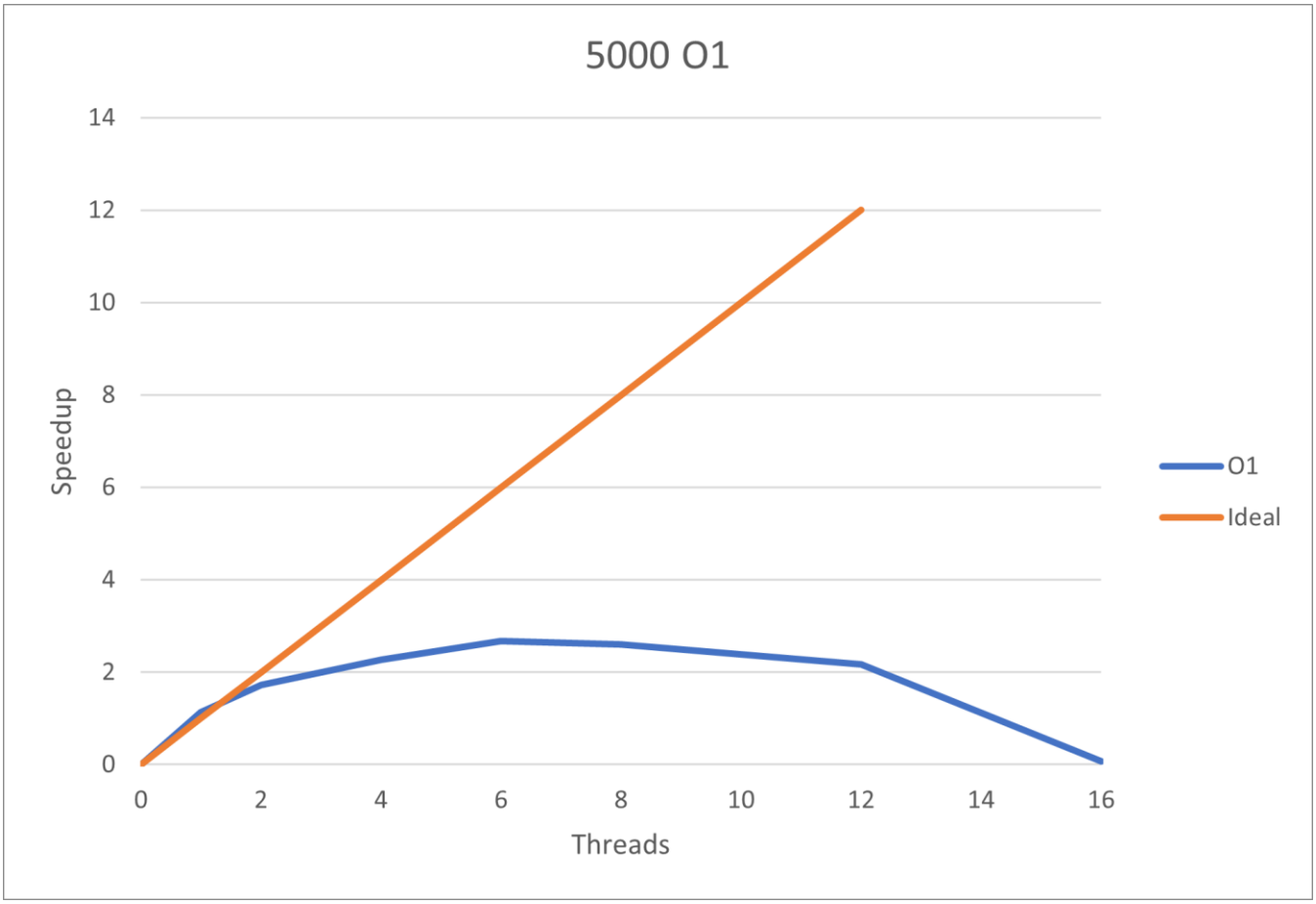
Misure O0-5000

Type	Time	Speedup
Sequential	*0,038996*	*/*
Threads 1	*0,055516*	*0,702435*
Threads 2	*0,035142*	*1,109672*
Threads 4	*0,022611*	*1,724626*
Threads 6	*0,016755*	*2,327398*
Threads 8	*0,014772*	*2,639935*
Threads 12	*0,013074*	*2,982696*
Threads 16	*0,090036*	*0,433114*



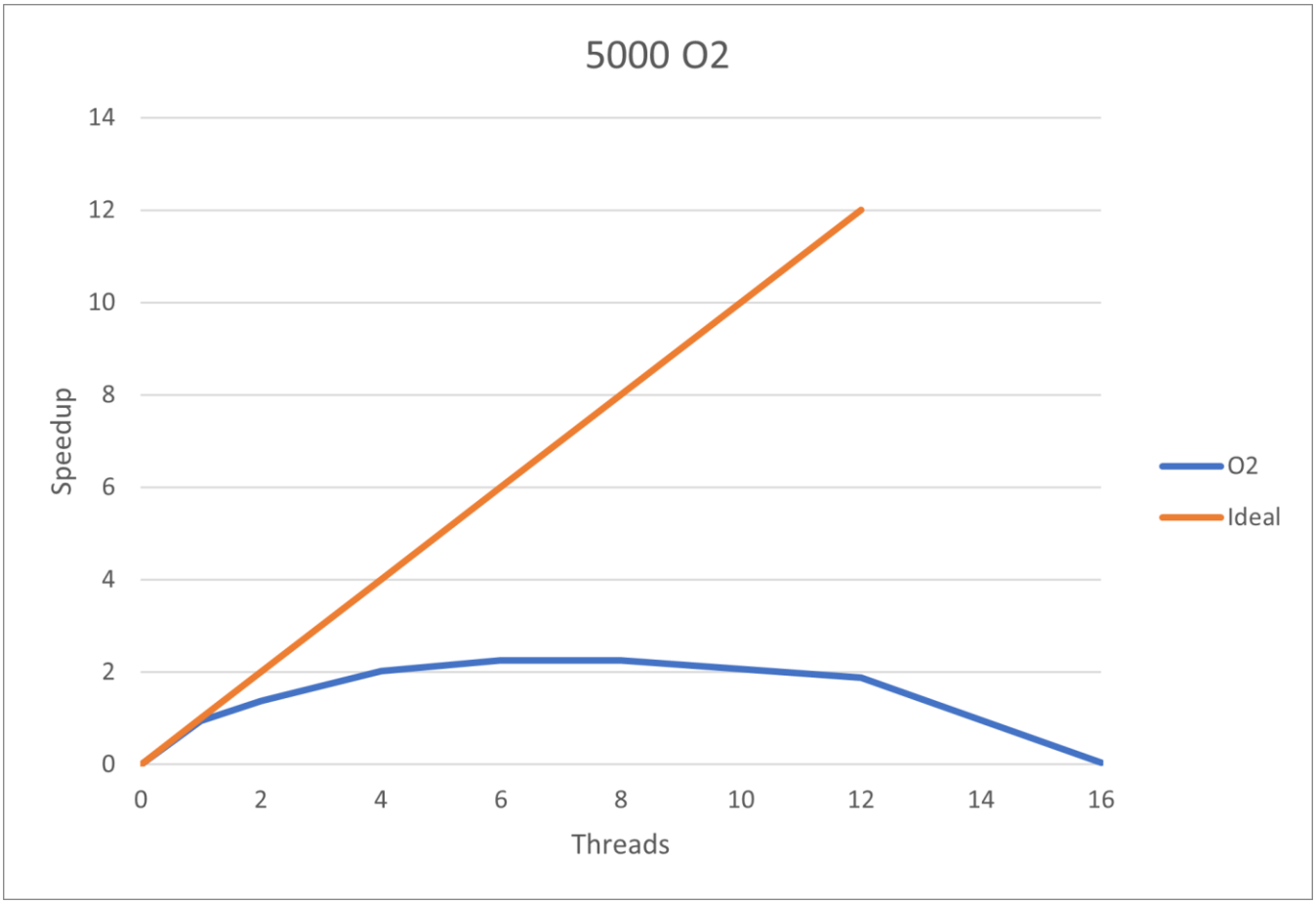
Misure O1-5000

Type	Time	Speedup
Sequential	*0,018931*	*/*
Threads 1	*0,016676*	*1,135219*
Threads 2	*0,011031*	*1,716121*
Threads 4	*0,008338*	*2,270496*
Threads 6	*0,007094*	*2,668675*
Threads 8	*0,007294*	*2,595484*
Threads 12	*0,008739*	*2,1662*
Threads 16	*0,274595*	*0,068941*



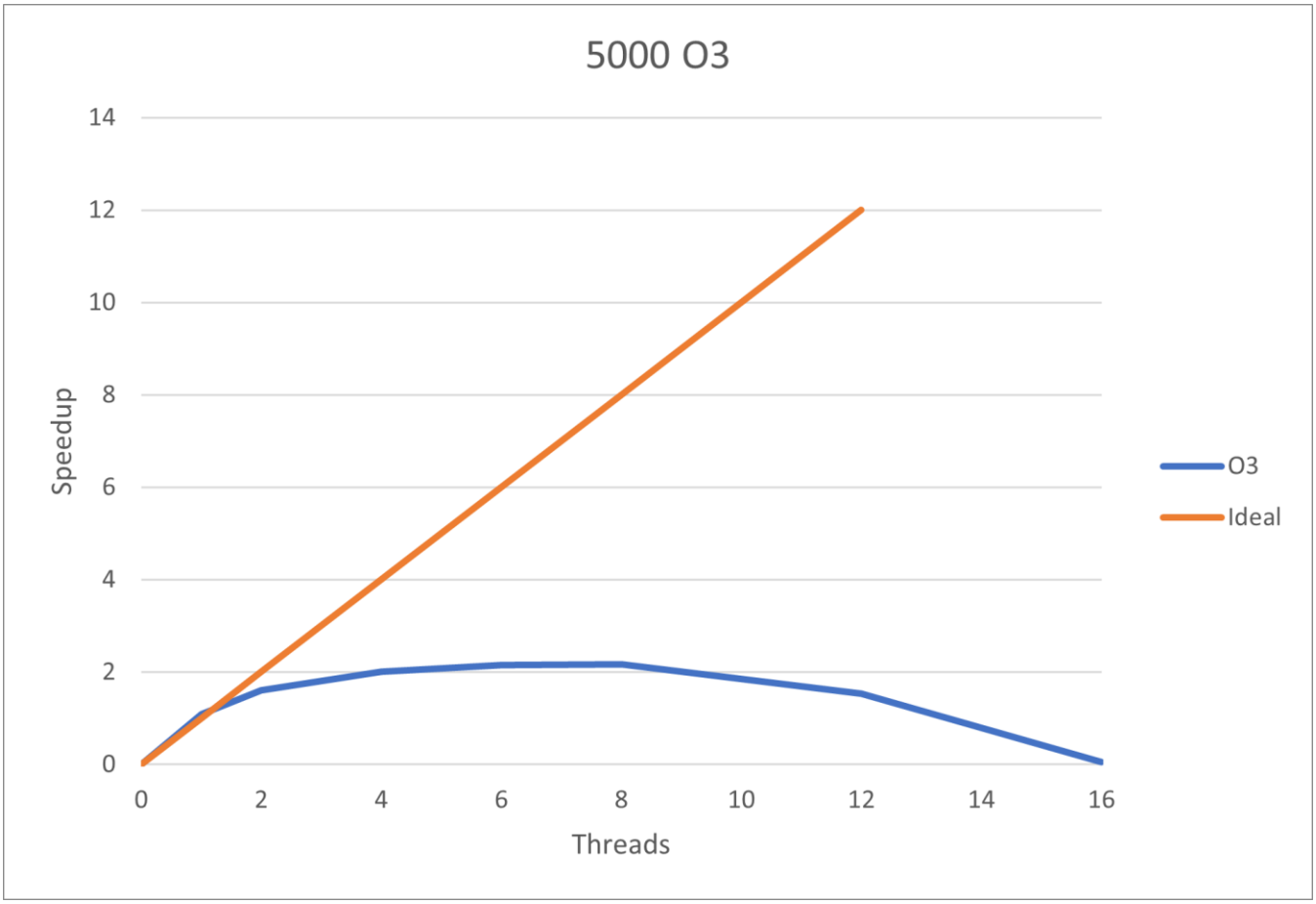
Misure O2-5000

Type	Time	Speedup
Sequential	*0,015922*	*/*
Threads 1	*0,016809*	*0,947255*
Threads 2	*0,011567*	*1,37652*
Threads 4	*0,007876*	*2,021587*
Threads 6	*0,007075*	*2,250394*
Threads 8	*0,007082*	*2,248272*
Threads 12	*0,00848*	*1,877598*
Threads 16	*0,382191*	*0,04166*



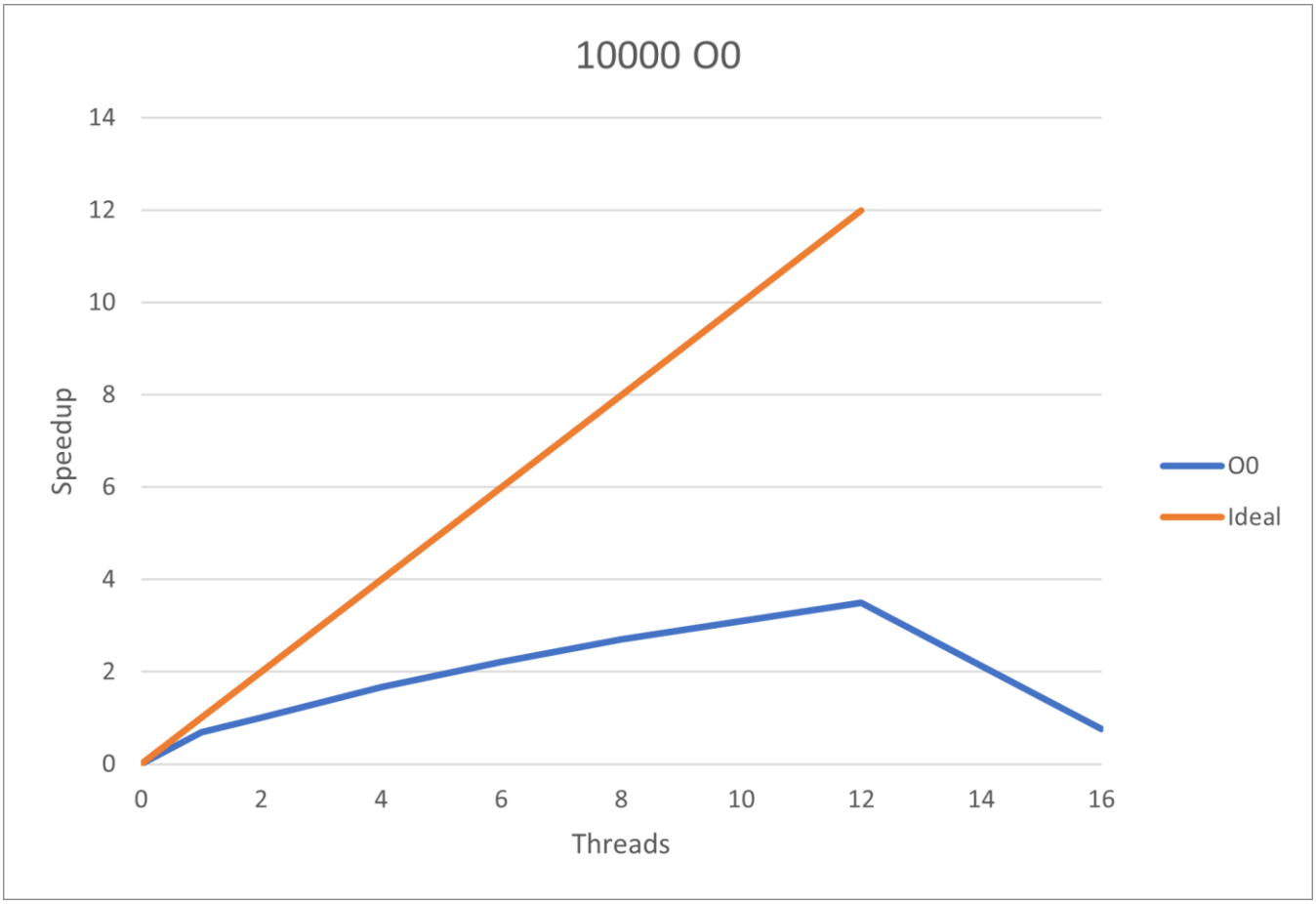
Misure O3-5000

Type	Time	Speedup
Sequential	*0,016373*	*/*
Threads 1	*0,014991*	*1,092218*
Threads 2	*0,010183*	*1,60792*
Threads 4	*0,008143*	*2,010662*
Threads 6	*0,007636*	*2,144287*
Threads 8	*0,007534*	*2,173382*
Threads 12	*0,010679*	*1,533257*
Threads 16	*0,37214*	*0,043998*



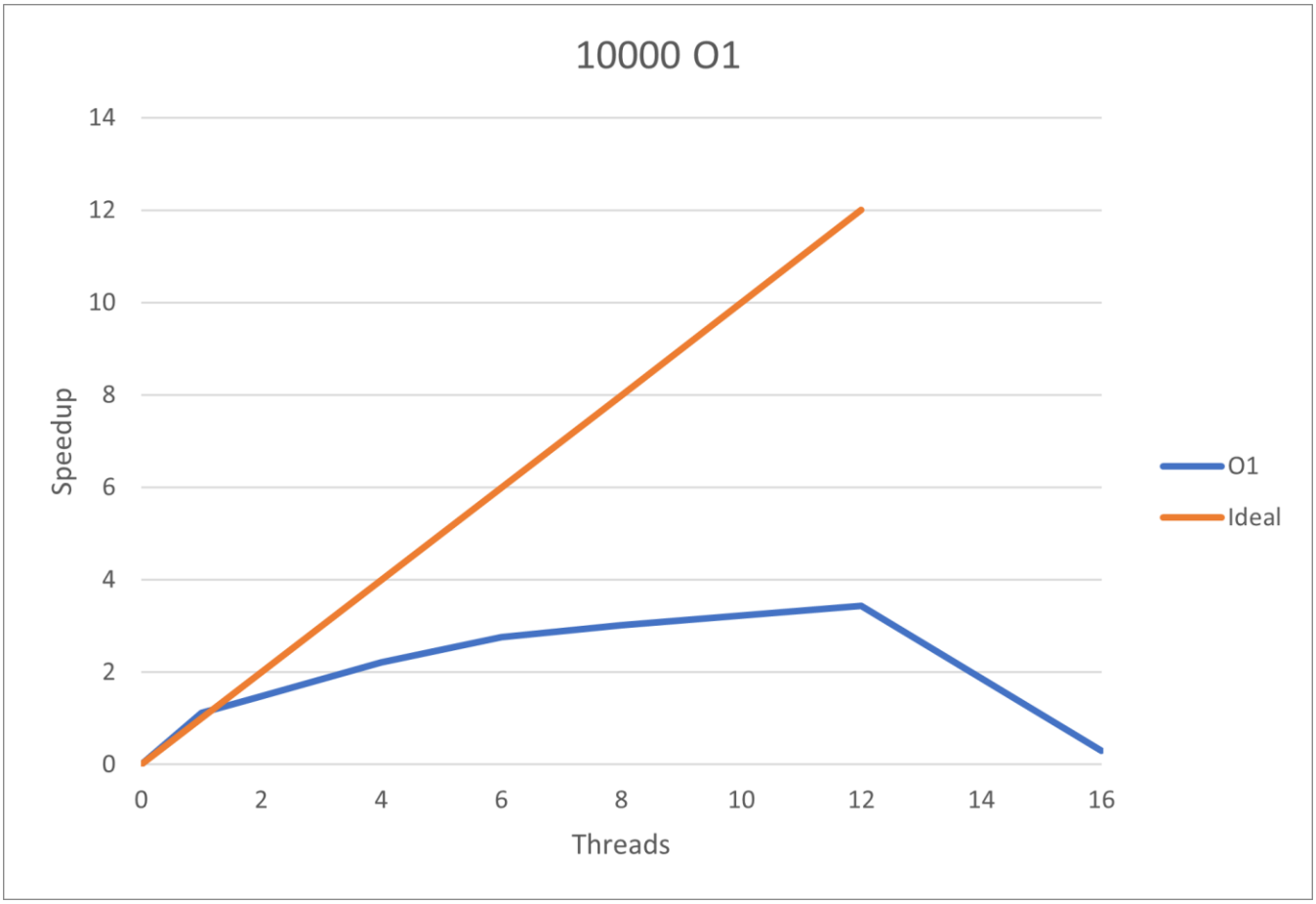
Misure O0-10000

Type	Time	Speedup
Sequential	*0,154713*	*/*
Threads 1	*0,222477*	*0,695412*
Threads 2	*0,154456*	*1,001663*
Threads 4	*0,093121*	*1,661411*
Threads 6	*0,069811*	*2,216178*
Threads 8	*0,057121*	*2,708491*
Threads 12	*0,044185*	*3,501488*
Threads 16	*0,20264*	*0,763486*



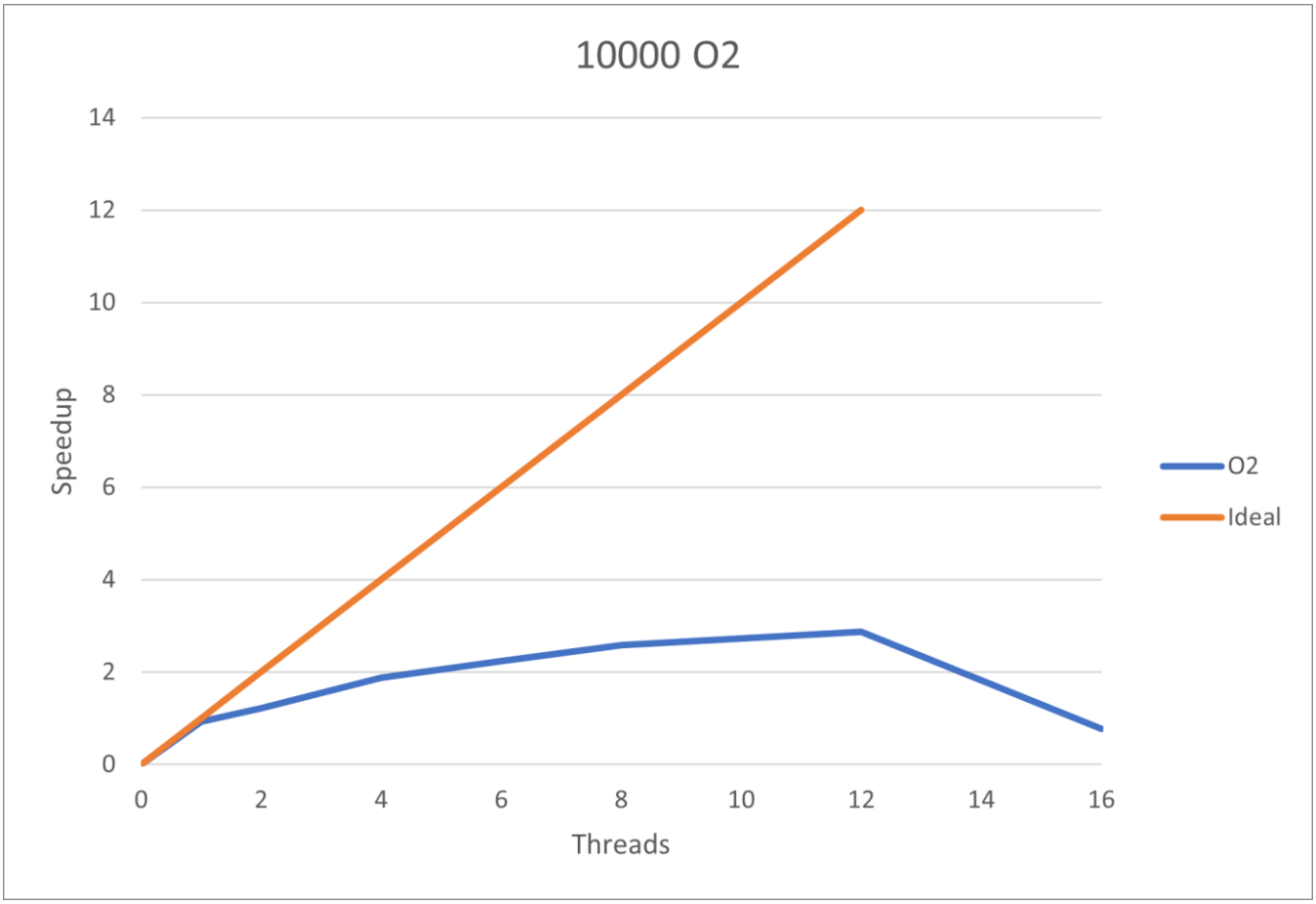
Misure O1-10000

Type	Time	Speedup
Sequential	*0,072632*	*/*
Threads 1	*0,065244*	*1,113237*
Threads 2	*0,048979*	*1,482913*
Threads 4	*0,032798*	*2,21451*
Threads 6	*0,026329*	*2,758591*
Threads 8	*0,024035*	*3,021924*
Threads 12	*0,02115*	*3,434067*
Threads 16	*0,237448*	*0,305885*



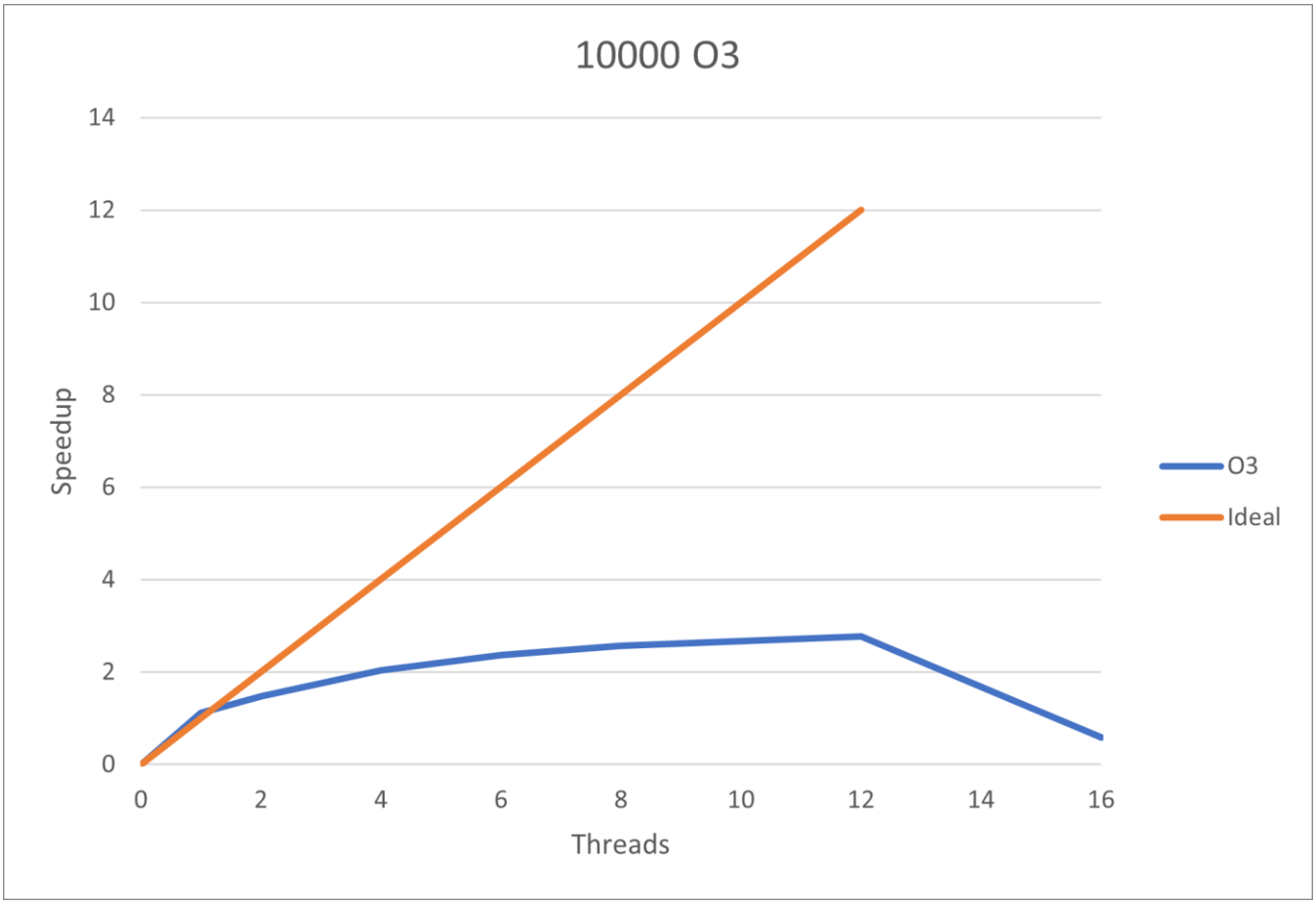
Misure O2-10000

Type	Time	Speedup
Sequential	*0,060134*	*/*
Threads 1	*0,064894*	*0,926642*
Threads 2	*0,049509*	*1,214606*
Threads 4	*0,032132*	*1,87147*
Threads 6	*0,026827*	*2,241558*
Threads 8	*0,023303*	*2,580531*
Threads 12	*0,020936*	*2,872291*
Threads 16	*0,07791*	*0,771837*



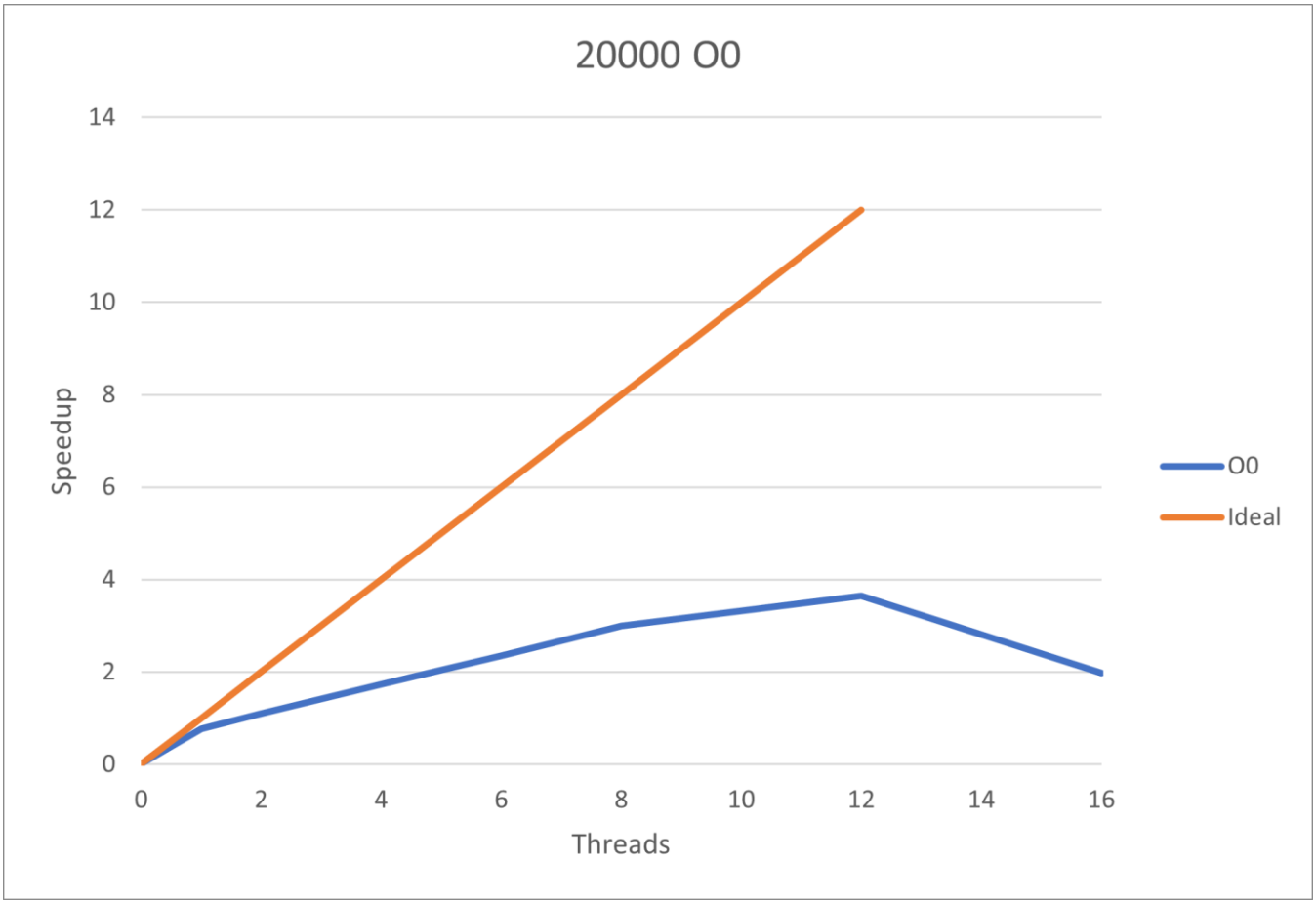
Misure O3-10000

Type	Time	Speedup
Sequential	*0,063369*	*/*
Threads 1	*0,057184*	*1,108175*
Threads 2	*0,042992*	*1,473999*
Threads 4	*0,031158*	*2,033783*
Threads 6	*0,026751*	*2,368908*
Threads 8	*0,024683*	*2,567323*
Threads 12	*0,02291*	*2,765963*
Threads 16	*0,10957*	*0,578346*



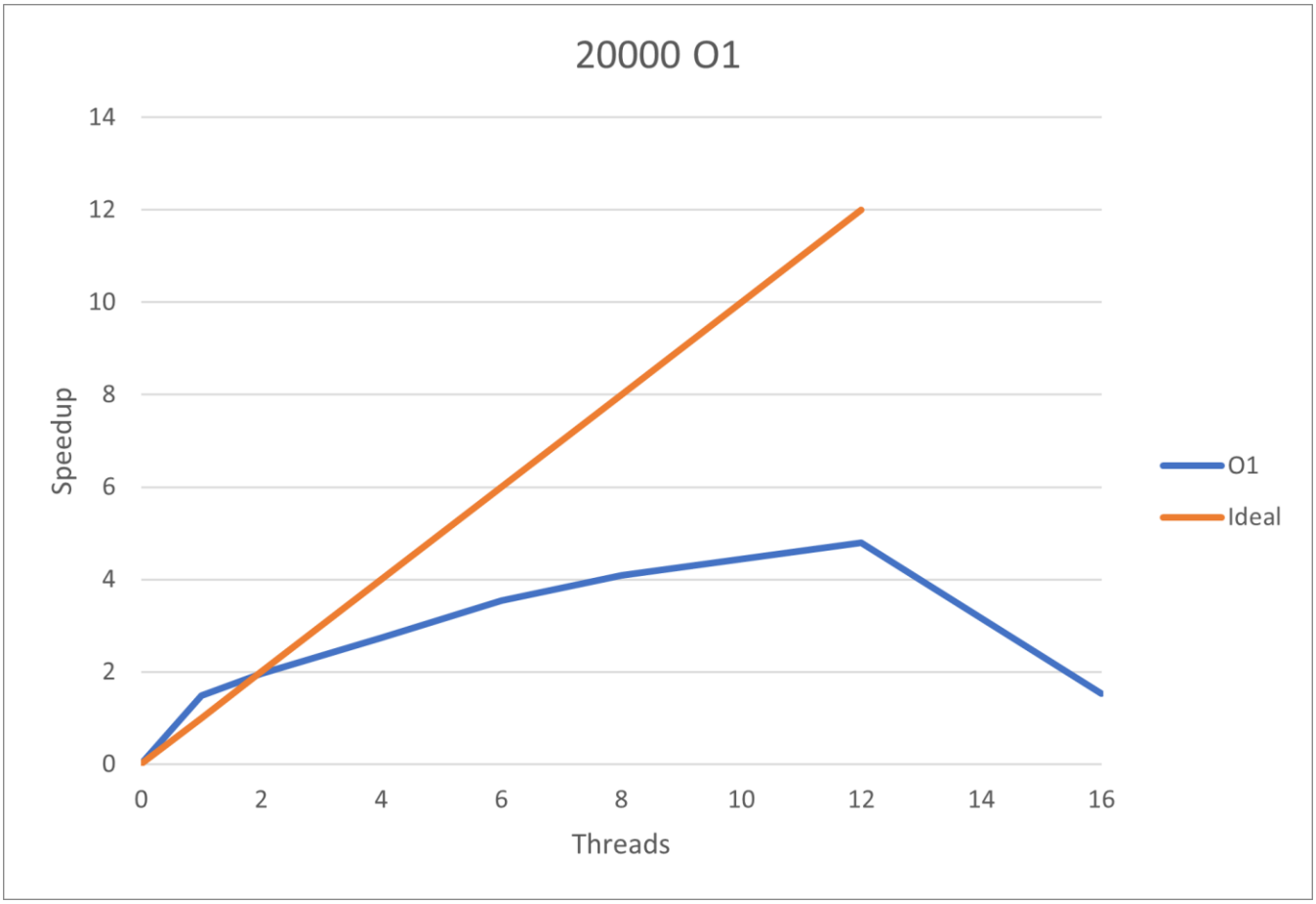
Misure O0-20000

Type	Time	Speedup
Sequential	*0,672526*	*/*
Threads 1	*0,878945*	*0,765151*
Threads 2	*0,609347*	*1,103682*
Threads 4	*0,388294*	*1,732*
Threads 6	*0,285734*	*2,353675*
Threads 8	*0,224746*	*2,992387*
Threads 12	*0,184368*	*3,647737*
Threads 16	*0,339336*	*1,981889*



Misure O1-20000

Type	Time	Speedup
Sequential	*0,372765*	*/*
Threads 1	*0,251144*	*1,484272*
Threads 2	*0,190604*	*1,955706*
Threads 4	*0,136383*	*2,733219*
Threads 6	*0,10538*	*3,537353*
Threads 8	*0,091231*	*4,085949*
Threads 12	*0,077859*	*4,787673*
Threads 16	*0,243761*	*1,529226*

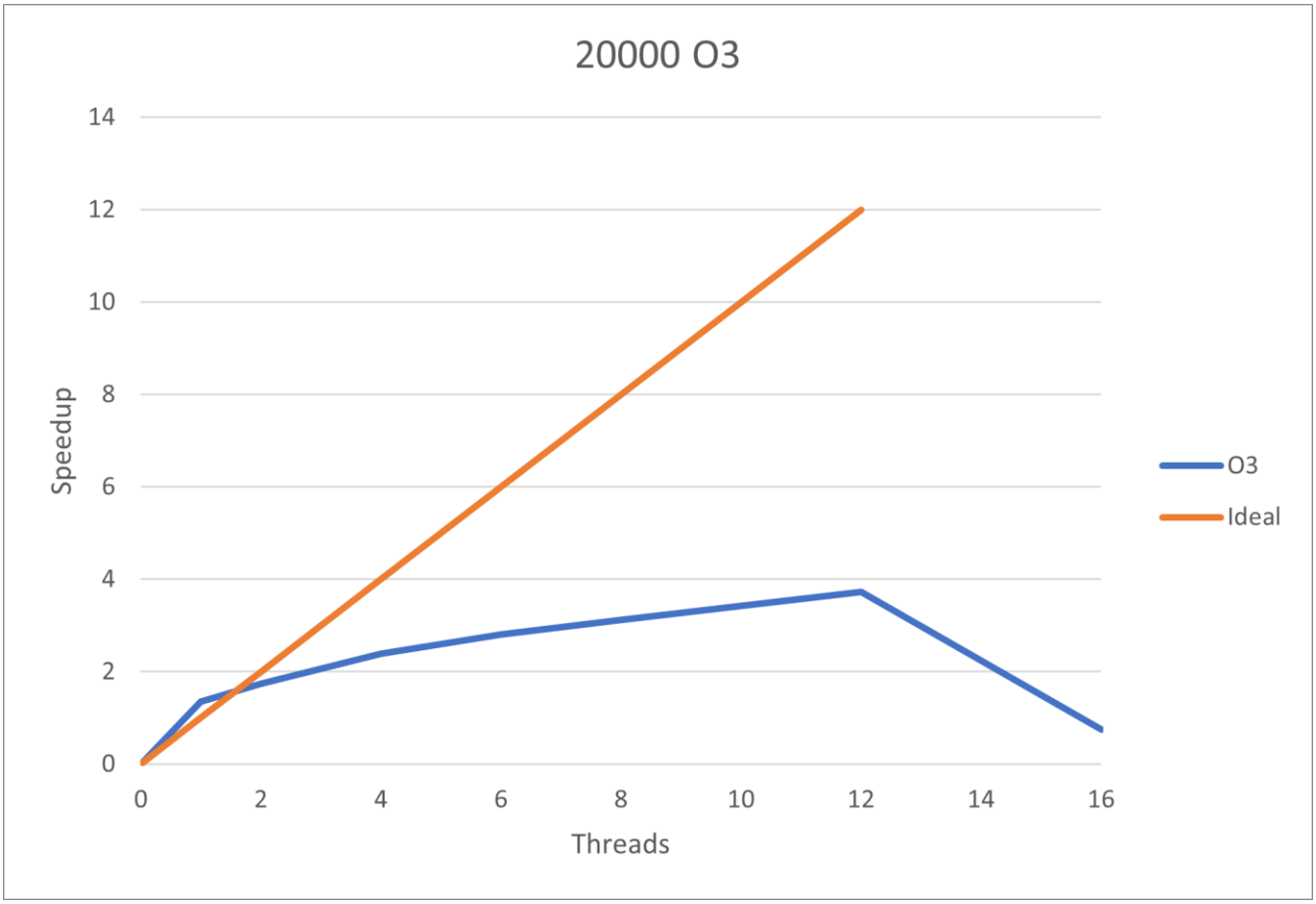


Misure O2-20000

Type	Time	Speedup
Sequential	*0,300408*	*/*
Threads 1	*0,253765*	*1,183804*
Threads 2	*0,196647*	*1,527648*
Threads 4	*0,132005*	*2,275728*
Threads 6	*0,104417*	*2,876999*
Threads 8	*0,090224*	*3,329582*
Threads 12	*0,075446*	*3,981754*
Threads 16	*0,296884*	*1,011869*

Misure O3-20000

Type	Time	Speedup
Sequential	*0,301723*	*/*
Threads 1	*0,222157*	*1,358153*
Threads 2	*0,173991*	*1,73413*
Threads 4	*0,126122*	*2,392318*
Threads 6	*0,107599*	*2,80413*
Threads 8	*0,096731*	*3,119191*
Threads 12	*0,080989*	*3,725476*
Threads 16	*0,402009*	*0,750537*



Conclusioni e considerazioni

L'algoritmo implementato risulta molto efficiente su grafi con molti vertici, con un andamento crescente fino a 12 threads, oltre questa soglia lo speedup tende a calare dovuto alla saturazione dei thread su singolo processore.

Nel caso in cui ci sono pochi vertici (come nel caso di 1000 vertici), non si notano molte differenze tra il sequenziale e il parallelo. Questo è dovuto al fatto che i tempi necessari affinché ci sia uno scambio di messaggi hanno un peso rilevante rispetto al mero tempo di esecuzione.

Man mano che il numero di vertici aumenta notiamo che lo speedup sale sempre di più.

È importante precisare che sono stati utilizzati al massimo 20000 vertici poiché la matrice di incidenza risulterebbe troppo elevata per la memoria utilizzata.

Inoltre, ogni BFS effettuata viene controllata e testata tramite un file di supporto contenente la BFS corretta.

L'ottimizzazione che presenta uno speedup maggione è O0 e O1, questo è dovuto principalmente all'abbassamento del tempo di esecuzione dell'algoritmo sequenziale.

Infatti, prendendo in considerazione le misurazioni su 10'000 vertici, abbiamo:

type	O0	O1	O2	O3
Sequential	*0,154713*	*0,072632*	*0,060134*	*0,063369*
Threads 12	*0,044185*	*0,02115*	*0,020936*	*0,02291*
Speedup	*3,501488*	*3,434067*	*2,872291*	*2,765963*

Da questa tabella si evince che entrambi gli algoritmi risultano migliorati nell'ottimizzazione. L'algoritmo parallelo si discosta di un nulla per le ottimizzazioni O1, O2 e O3, mentre quello sequenziale ha un tempo di esecuzione maggiore per l'ottimizzazione O1, il quale si abbassa, invece, per le ottimizzazioni O2 e O3.

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_MPI”, 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.