BINARIZZAZIONE DI UNA MATRICE

PROGETTO - SISTEMI CONCORRENTI E PARALLELI M

Gabriele Doti
0001245897

PROPOSTA DI PROGETTO

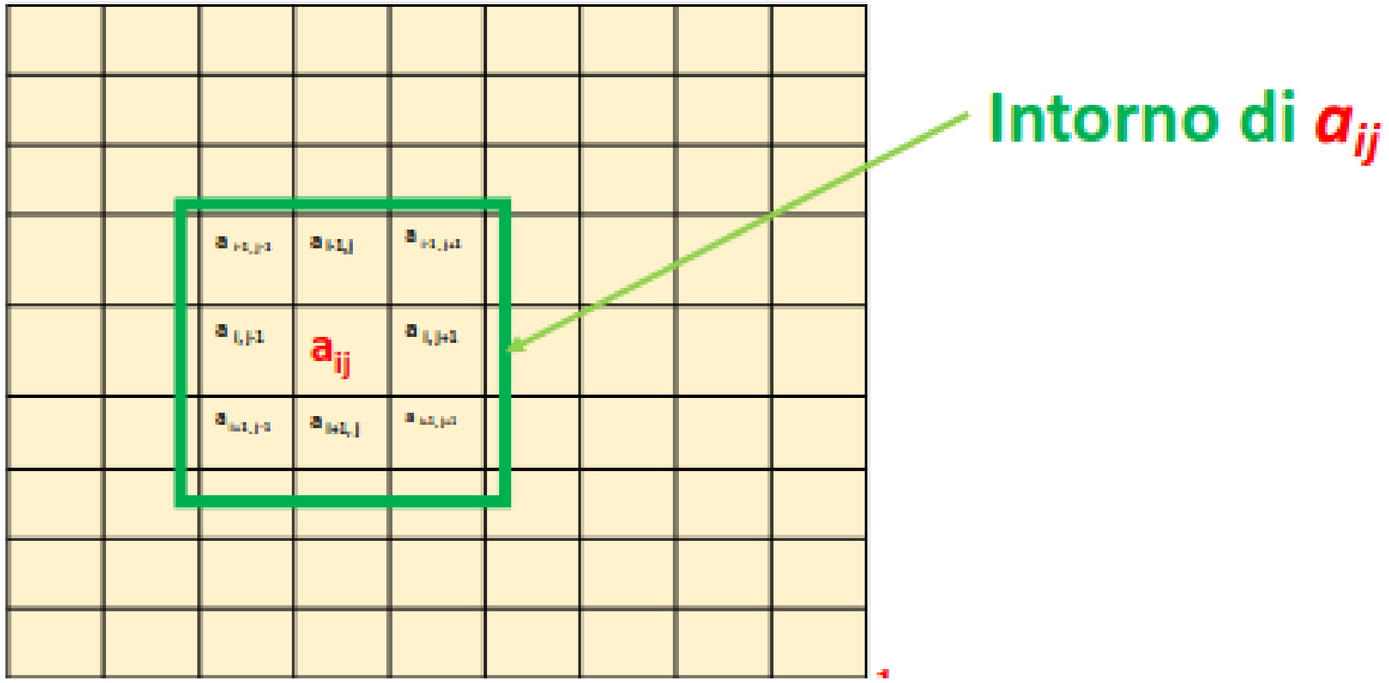
OBIETTIVI

Scopo dell'algoritmo

L'obiettivo è convertire una matrice quadrata di numeri reali in una matrice binaria, decidendo il valore di ogni elemento in base al confronto con la media dei suoi vicini.

- **Matrice di input (A)** È una matrice quadrata di dimensione N×N, con N≥2000, contenente valori reali.
- **Matrice di output (T)** È una matrice quadrata della stessa dimensione N×N, i cui elementi assumono esclusivamente valori binari {0,1}.

Definiamo «intorno» di aij la sottomatrice di A di dimensione 3X3 che ha aij nella posizione centrale.



O B I E T T I V I

Funzionamento

1. Calcolo della media locale

1. Calcolo della media locale
Per ogni posizione (i,j), si considera l'intorno I_{ij} formato dall'elemento stesso e dai suoi otto vicini immediati (finestra 3×3).
La media aritmetica è definita come:

$$m_{i,j} = \frac{1}{9} \sum_{x=i-1}^{i+1} \sum_{y=j-1}^{j+1} a_{x,y}$$

Quindi ogni elemento richiede 9 accessi in memoria e 8 somme + 1 divisione. Questo dettaglio diventa rilevante quando parli di prestazioni.

2. Sogliatura binaria

Una volta calcolata la media locale, se il valore centrale è maggiore della media dei vicini:

$t_{ij}=1$ altrimenti $t_{ij}=0$

IMPLEMENTAZIONE SERIALE

ALGORITMO

L'algoritmo seriale analizza ogni cella $A(i, j)$, calcola poi la media dei vicini e di sé stessa e assegna un valore binario basato sul confronto tra il valore centrale e la media locale.

```
for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
        double sum = 0.0;
        int count = 0;

        // Loop over neighboring cells
        for(int di=-1;di<=1;di++){
            for(int dj=-1;dj<=1;dj++){
                int ni = i + di;
                int nj = j + dj;
                if(ni >= 0 && ni < N && nj >= 0 && nj < N){
                    sum += A[ni*N + nj];      // matrice memorizzata come blocco contiguo di memoria
                    count++;
                }
            }
        }

        T[i*N + j] = (A[i*N + j] * count > sum) ? 1 : 0;     // ottimizzazione
    }
}
```

IMPLEMENTAZIONE PARALLELA: OMP

ALGORITMO

L'algoritmo seriale analizza ogni cella $A(i, j)$, calcola poi la media dei vicini e di sé stessa e assegna un valore binario basato sul confronto tra il valore centrale e la media locale.

```
#pragma omp parallel for schedule(static) num_threads(P)
for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
        double sum = 0.0;
        int count = 0;

        // Loop over neighbors
        for(int di=-1;di<=1;di++){
            for(int dj=-1;dj<=1;dj++){
                int ni = i + di;
                int nj = j + dj;
                if(ni >= 0 && ni < N && nj >= 0 && nj < N){
                    sum += A[ni*N + nj];
                    count++;
                }
            }
        }

        T[i*N + j] = (A[i*N + j] * count > sum) ? 1 : 0;
    }
}
```

Possibili approcci:

1. schedule(static) ↠ *Bilanciato, basso overhead*
   - Thread 0 ↠ i = 0, 1, 2, 3
   - Thread 1 ↠ i = 4, 5, 6, 7
   - Thread 2 ↠ i = 8, 9, 10

2. schedule(static, 2) ↠ *Meno bilanciato, basso overhead*
   - Thread 0 ↠ i = [0, 1], [6, 7]
   - Thread 1 ↠ i = [2, 3], [8, 9]
   - Thread 2 ↠ i = [4, 5], 10

3. schedule(dynamic, 2) ↠ *Bilanciato, alto overhead*
   - Thread 0 ↠ i = 0, 1
   - Thread 1 ↠ i = 2, 3
   - Thread 2 ↠ i = 4, 5

↠ *6. 7. 8. 9. 10 rimangono in coda*

STRONG SCALING

I tempi mostrati corrispondono alla media di **2 rilevamenti indipendenti**

Lo **speedup teorico** è determinato mediante la formula: $S_p = \frac{T_1}{T_p}$

L'**efficiency** è determinato mediante la formula: $E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p}$

P	N	Mean Time [s]	Speedup	Efficiency
1	10000	0.58602	1.00	1.00
2	10000	0.29317	2.00	1.00
4	10000	0.14659	3.99	1.00
8	10000	0.07357	7.97	0.996
12	10000	0.04925	11.89	0.99
16	10000	0.03717	15.77	0.985
20	10000	0.02989	19.61	0.981
24	10000	0.02525	23.21	0.967
48	10000	0.01390	42.15	0.878

Performance results using uint16_t

P	N	Mean Time [s]	Speedup	Efficiency
1	10000	0.71553	1.00	1.00
2	10000	0.35802	2.00	1.00
4	10000	0.17923	3.99	1.00
8	10000	0.08979	7.97	1.00
12	10000	0.06014	11.90	0.99
16	10000	0.04528	15.80	0.99
20	10000	0.03645	19.63	0.98
24	10000	0.03229	22.17	0.92
48	10000	0.02368	30.23	0.63

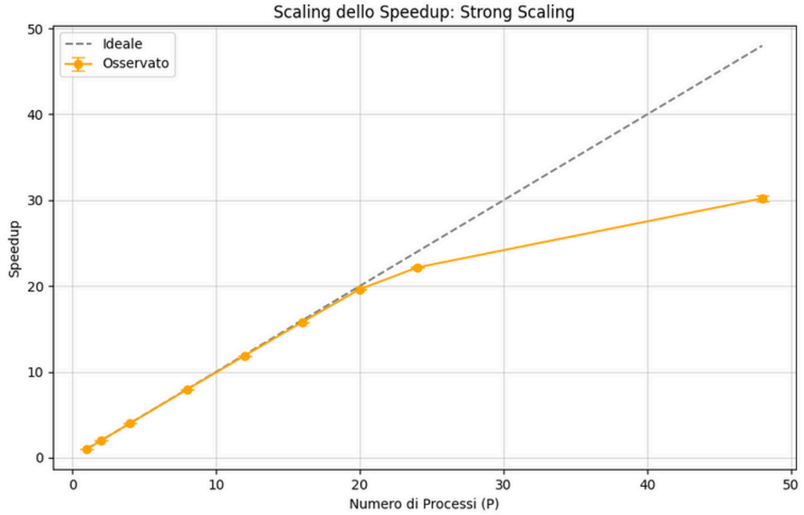
Performance results using double

P	N	Mean Time [s]	Speedup	Efficiency
1	10000	0.58485	1.00	1.00
2	10000	0.29292	2.00	1.00
4	10000	0.14655	3.99	1.00
8	10000	0.07347	7.96	0.99
12	10000	0.04923	11.88	0.99
16	10000	0.03713	15.75	0.98
20	10000	0.02993	19.54	0.98
24	10000	0.02720	21.51	0.90
48	10000	0.01764	33.16	0.69

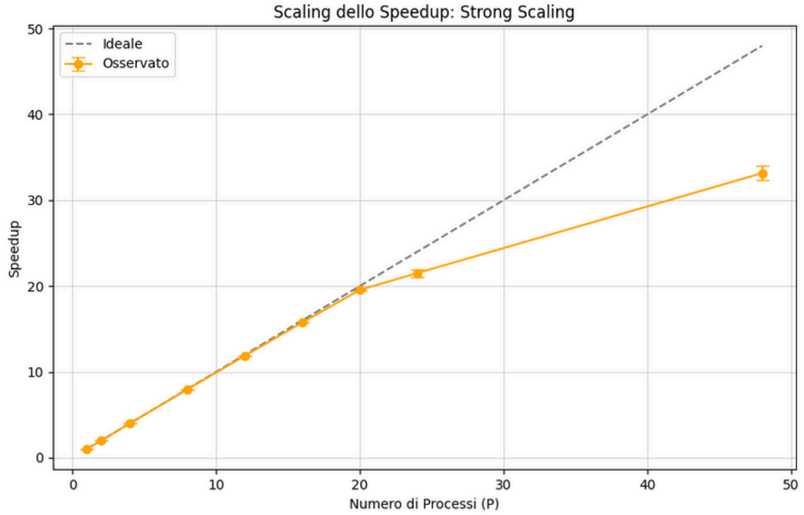
Performance results using float

STRONG SCALING

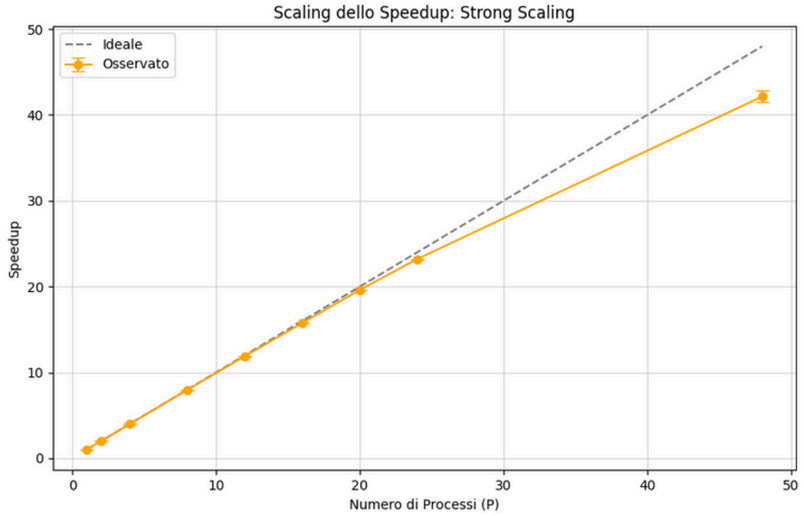
double:



float:



uint16_t:



- La **dimensione** della matrice rimane **costante** mentre **aumenta il numero di thread o nodi**.
- Il **lavoro totale** da eseguire rimane **invariato**.
- Il **lavoro** assegnato a ciascun **nodo**/**thread diminuisce al crescere del numero di nodi**.
- Questo scenario è tipico per studiare la **scalabilità forte** di un algoritmo.
- La **legge di Amdahl** fornisce un **limite teorico** allo **speedup** ottenibile a causa della frazione seriale del codice.

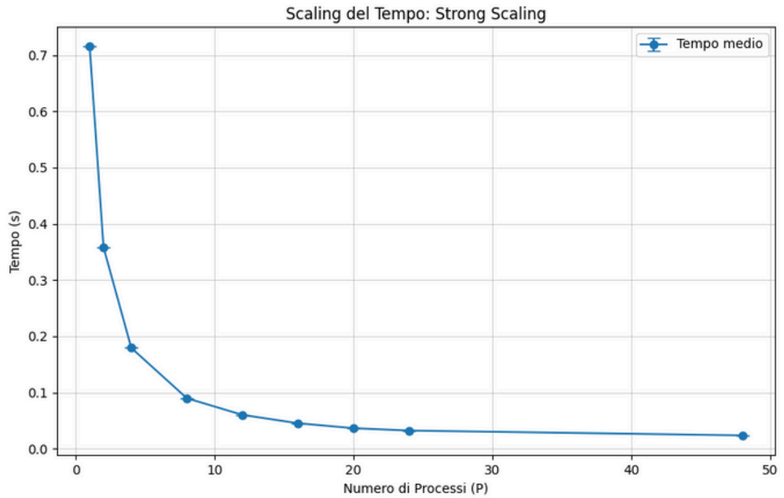
$$S_p = \frac{1}{(1-f) + \frac{f}{p}}$$

dove:

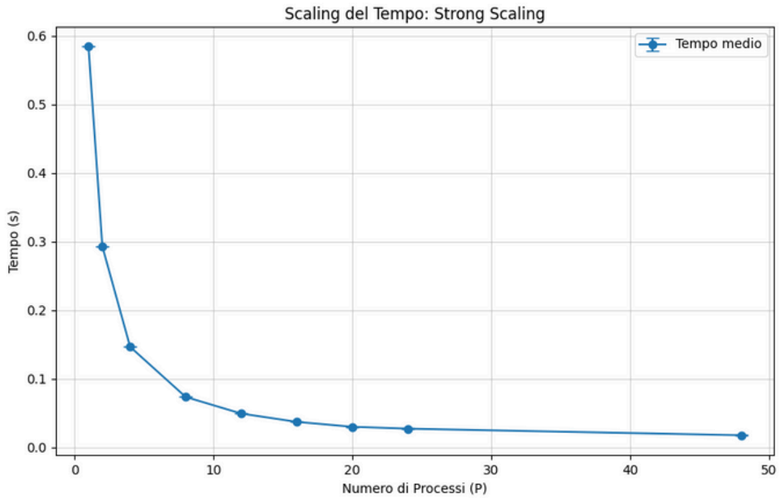
- f è la frazione del codice parallelizzabile,
- $(1-f)$ è la frazione seriale,
- p è il numero di processori/thread.

STRONG SCALING

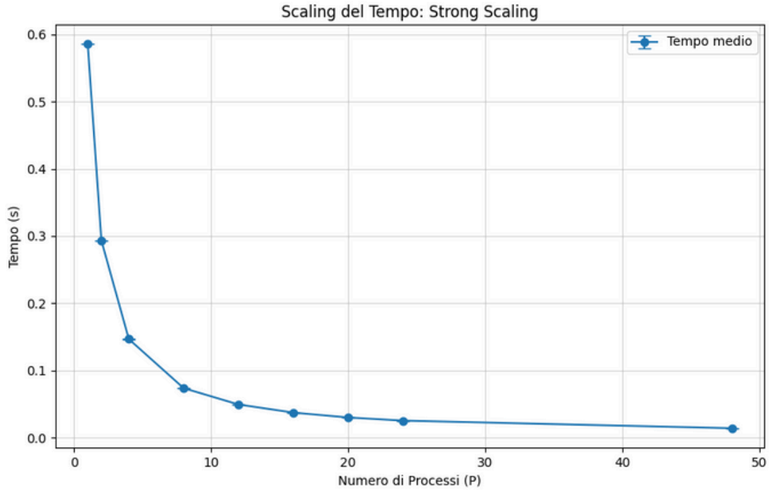
double:



float:



uint16_t:



- La **dimensione** della matrice rimane **costante** mentre **aumenta il numero di thread o nodi**.
- Il **lavoro totale** da eseguire rimane **invariato**.
- Il **lavoro** assegnato a ciascun **nodo**/**thread diminuisce al crescere del numero di nodi**.
- Questo scenario è tipico per studiare la **scalabilità forte** di un algoritmo.
- La **legge di Amdahl** fornisce un **limite teorico** allo **speedup** ottenibile a causa della frazione seriale del codice.

$$S_p = \frac{1}{(1-f) + \frac{f}{p}}$$

dove:

- f è la frazione del codice parallelizzabile,
- $(1-f)$ è la frazione seriale,
- p è il numero di processori/thread.

WEAK SCALING

I tempi mostrati corrispondono alla media di **2 rilevamenti indipendenti**

Lo **speedup teorico** è determinato mediante la formula:

$$S_p^{\text{scaled}} = P \cdot \frac{T_1}{T_p}$$

L'**efficiency** è determinato mediante la formula:

$$E_p^{\text{scaled}} = \frac{S_p^{\text{scaled}}}{P} = \frac{T_1}{T_p}$$

P	N	Mean Time [s]	Scaled Speedup	Scaled Efficiency
1	2000	0.023565	1.000	1.000
2	2828	0.0235055	1.997	0.9985
4	4000	0.023765	3.832	0.958
8	5656	0.0237525	7.902	0.988
12	6924	0.023841	11.471	0.956
16	8000	0.0241265	15.025	0.940
20	8940	0.0240855	18.904	0.945
24	9792	0.024426	22.079	0.920
48	13872	0.025081	42.866	0.893

Performance using uint16_t

P	N	Mean Time [s]	Scaled Speedup	Scaled Efficiency
1	2000	0.028662	1.000	1.000
2	2828	0.028754	1.992	0.996
4	4000	0.028822	3.974	0.994
8	5656	0.028973	7.905	0.988
12	6924	0.0289995	11.872	0.989
16	8000	0.0292025	15.694	0.981
20	8940	0.030005	19.102	0.955
24	9792	0.031495	21.831	0.910
48	13872	0.046001	29.936	0.624

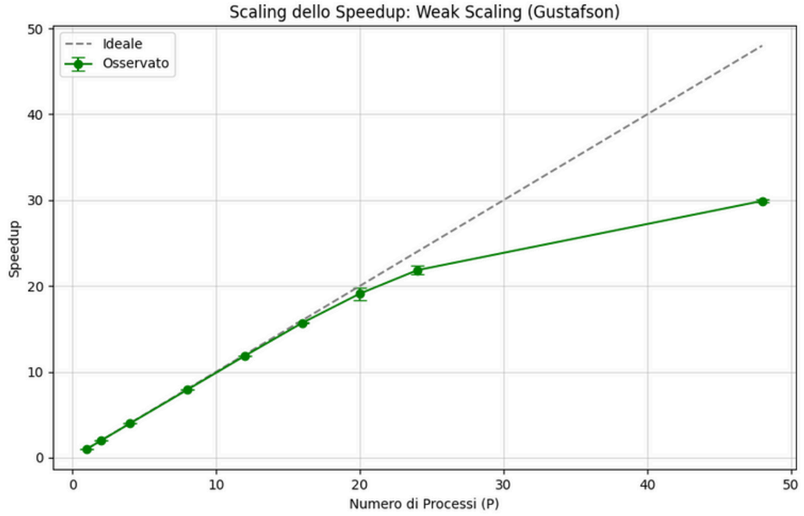
Performance results using double

P	N	Mean Time [s]	Scaled Speedup	Scaled Efficiency
1	2000	0.023465	1.000	1.000
2	2828	0.023501	1.995	0.998
4	4000	0.023615	3.971	0.993
8	5656	0.0237205	7.917	0.990
12	6924	0.023823	11.836	0.986
16	8000	0.024019	15.675	0.980
20	8940	0.0240205	19.530	0.977
24	9792	0.025810	21.849	0.911
48	13872	0.033467	31.857	0.664

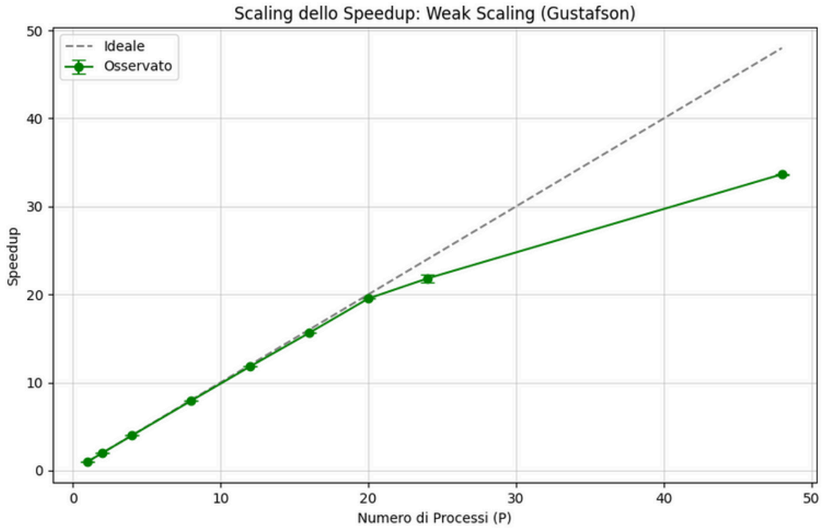
Performance results using float

WEAK SCALING

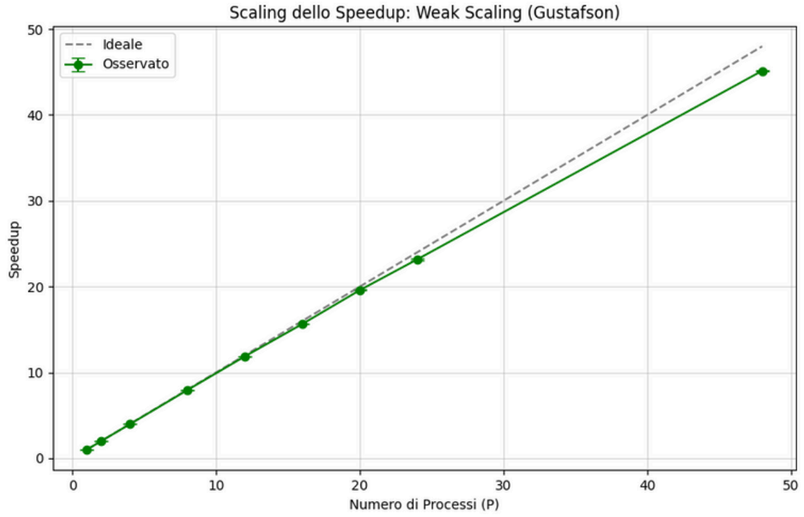
double:



float:



uint16_t:



- Si mantiene **costante il carico di lavoro per singolo nodo/thread.**
- La **dimensione totale** del problema **aumenta proporzionalmente** al numero di nodi.
- Questo scenario studia la scalabilità debole dell'algoritmo.
- La **legge di Gustafson** permette di prevedere lo **speedup teorico** quando il problema cresce con il numero di processori.

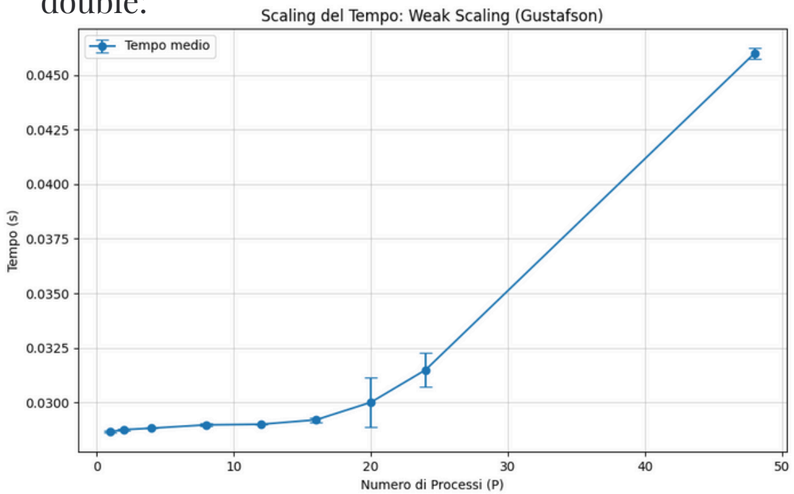
$$S_p = p - (1 - f) \cdot (p - 1)$$

dove:

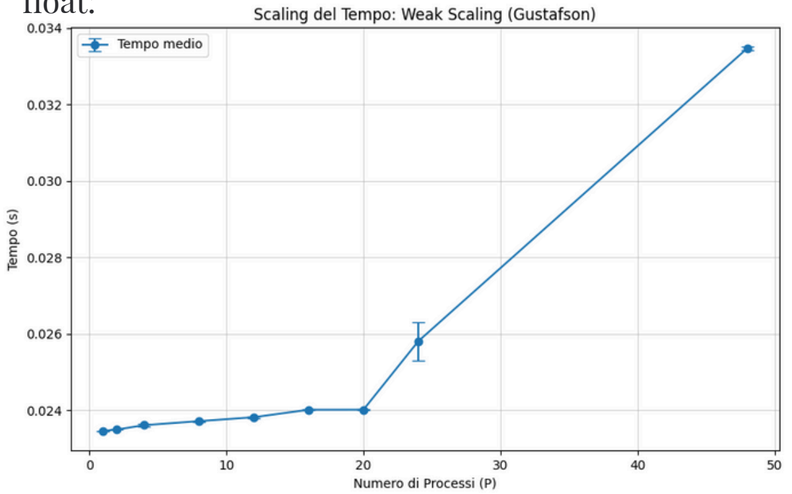
- f è la frazione del codice parallelizzabile,
- $(1 - f)$ è la frazione seriale,
- p è il numero di processori/thread.

WEAK SCALING

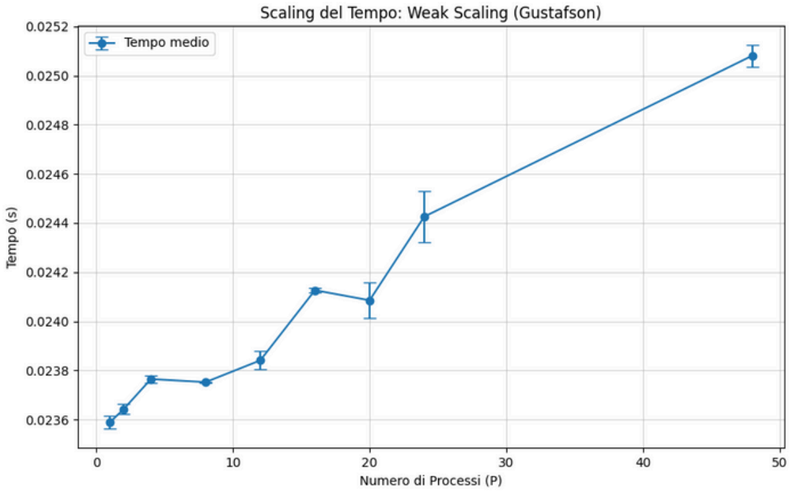
double:



float:



uint16_t:



- Si mantiene **costante il carico di lavoro per singolo nodo/thread.**
- La **dimensione totale** del problema **aumenta proporzionalmente** al numero di nodi.
- Questo scenario studia la scalabilità debole dell'algoritmo.
- La **legge di Gustafson** permette di prevedere lo **speedup teorico** quando il problema cresce con il numero di processori.

$$S_p = p - (1 - f) \cdot (p - 1)$$

dove:

- f è la frazione del codice parallelizzabile,
- $(1 - f)$ è la frazione seriale,
- p è il numero di processori/thread.

IMPLEMENTAZIONE PARALLELA: MPI

M P I

Distribuzione del carico di lavoro

Dal momento che la dimensione della matrice N non sempre si divide esattamente per il numero di processi size, è necessario adottare una strategia di suddivisione dei dati che bilanci il carico di lavoro tra i processi.
Il nodo master utilizza una logica basata su:

- divisione intera: ogni processo riceve almeno $\lfloor N/P \rfloor$ righe,
- resto: le righe rimaste vengono distribuite tra i primi processi per evitare squilibri.

```
int base_rows = N / size; // rows per process
int extra = N % size; // extra rows
```

Thread 0 ↠ i = 0, 1, 2, 3
Thread 1 ↠ i = 4, 5, 6, 7
Thread 2 ↠ i = 8, 9, 10

Possibili approcci

1 – Master invia righe + halo

- Il master (rank 0) distribuisce tutte le righe ai processi.
- Ogni processo riceve:
  - Le righe di sua competenza
  - Le righe halo necessarie per calcolare la media locale.
- Attenzione ai bordi:
  - Il primo processo riceve solo l'halo sotto
  - L'ultimo processo riceve solo l'halo sopra

2 – Scambio halo tra processi (Send/Recv)

- Ogni processo calcola le proprie righe senza passare dal master.
- Gli halo vengono scambiati solo tra processi adiacenti:
  - Un processo invia la riga superiore al processo precedente e quella inferiore al processo successivo.
  - Riceve allo stesso modo gli halo necessari.

APPROCCIO 1 - MASTER INVIA TUTTE LE RIGHE

- Il **master** (rank 0) è responsabile della **distribuzione dei dati** a tutti i processi.
- Per ogni processo, il master invia:
  - Le **righe assegnate** al processo
  - Le righe halo necessarie per calcolare la **media locale** (di solito la riga sopra e quella sotto ↠ halo)
- Attenzione **agli halo ai bordi**:
  - Il primo processo non ha righe sopra ↠ invia solo la riga sotto come halo
  - L'ultimo processo non ha righe sotto ↠ invia solo la riga sopra come halo
    ↠ Ogni processo riceve quindi solo le righe di sua competenza più gli halo necessari

Vantaggi:
- Chiarezza e semplicità nell'invio dei dati
- Ogni processo ha tutto ciò che serve per calcolare la sua porzione

Svantaggi:
- Il master può diventare un **collo di bottiglia** se la matrice è molto grande
- Maggior **overhead di comunicazione** se i dati non sono suddivisi uniformemente

APPROCCIO 1 - MASTER INVIA TUTTE LE RIGHE

Procedimento

1. Il processo Master (rank = 0) distuibuisce le righe + halo

```
if (my_rank == 0) {
    MPI_Scatterv(
        A,
        sendcounts,
        displs_scatterv,
        MPI_DOUBLE,
        recvbuf,
        sendcounts[0],
        MPI_DOUBLE,
        0,
        MPI_COMM_WORLD
    );
}
```

```
int MPI_Scatterv(
    const void *sendbuf,   // buffer del root
    const int *sendcounts, // numero di elementi
                              da inviare a ciascun
                              processo
    const int *displs,     // offset di partenza per
                              ogni blocco di sendbuf
    MPI_Datatype sendtype, // tipo di el. inviati
    void *recvbuf,         // buffer locale processo
    int recvcount,         // num elementi che p
                              riceve
    MPI_Datatype recvtype, // tipo di el. ricevuti
    int root,              // rank del processo che invia
    MPI_Comm comm          // comunicatore
);
```

APPROCCIO 1 - MASTER INVIA TUTTE LE RIGHE

Procedimento

2. Ogni processo riceve righe + halo e calcola i risultati

```
// Computer local sum
for (int i = first_row; i < last_row + 1; i++){ // for each row
    for (int j = 0; j < N; j++){ // for each column
        double sum = 0.0;
        int count = 0;

        // Loop over the 3x3 neighborhood
        for (int di = -1; di <= 1; di++){
            for (int dj = -1; dj <= 1; dj++){
                int ni = i + di;
                int nj = j + dj;
                if (ni >= 0 && ni < my_sendcount/N && nj >= 0 && nj < N){
                    sum += recvbuf[ni * N + nj];
                    count++;
                }
            }
        }

        sendbuf[(i - first_row) * N + j] = (recvbuf[i * N + j] * count > sum) ? 1 : 0; // ottimizzazione
    }
}
```

Processo con rank ≠ 0 ∧ rank ≠ N-1

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & \ldots\ldots\ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & \ldots\ldots\ldots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & \ldots\ldots\ldots & a_{3n} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & \ldots\ldots\ldots & a_{4n} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & \ldots\ldots\ldots & a_{5n} \\ .. & .. & .. & .. & .. & \ldots\ldots\ldots & .. \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & a_{n5} & \ldots\ldots\ldots & a_{nn} \end{pmatrix}$$

Processo con rank = 0

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & \ldots\ldots\ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & \ldots\ldots\ldots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & \ldots\ldots\ldots & a_{3n} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & \ldots\ldots\ldots & a_{4n} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & \ldots\ldots\ldots & a_{5n} \\ .. & .. & .. & .. & .. & \ldots\ldots\ldots & .. \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & a_{n5} & \ldots\ldots\ldots & a_{nn} \end{pmatrix}$$

APPROCCIO 1 - MASTER INVIA TUTTE LE RIGHE

Procedimento

3. Ogni processo invia righe calcolare (senza halo) e si forma la matrice risultato T

```
MPI_Gatherv(
    sendbuf,
    recvcounts[0],
    MPI_INT,
    T,
    recvcounts,
    displs_gatherv,
    MPI_INT,
    0,
    MPI_COMM_WORLD
);
```

```
int MPI_Gatherv(
    const void *sendbuf,   // buffer locale
                              del processo
    int sendcount,         // elementi da inviare
    MPI_Datatype sendtype, // tipo degli elementi
    void *recvbuf,         // buffer del root
    const int *recvcounts, // num di elementi
                              attesi da ciascun p
    const int *displs,     // offset di inizio per
                              ciascun blocco nel buffer
    MPI_Datatype recvtype, // tipo degli  elementi
                              ricevuti
    int root,              // rank del root
    MPI_Comm comm          // comunicatore
);
```

APPROCCIO 2 - SCAMBIO DI HALO TRA PROCESSI

- Il calcolo è **distribuito tra i processi**, che gestiscono autonomamente le loro righe assegnate.
- Gli **halo** necessari per il calcolo della media locale vengono **scambiati solo tra processi** adiacenti:
  - Ogni processo invia la **riga superiore al processo precedente** e **quella inferiore al processo successivo**.
  - Allo stesso modo **riceve le righe halo dai vicini**.
  - I bordi della matrice (primo e ultimo processo) inviano/ ricevono solo l'halo disponibile.
  - ↠ In questo modo, **ogni processo ha le righe necessarie** e gli halo per eseguire i calcoli senza passare dal master.

Vantaggi:

- **Riduce il collo di bottiglia** del master.
- Minore traffico complessivo di dati.

Svantaggi:

- **Implementazione più complessa.**
- Necessità di gestire correttamente la sincronizzazione per evitare deadlock.

APPROCCIO 2 - SCAMBIO DI HALO TRA PROCESSI

Funzioni per l'invio e la ricezione:

1. **SSend / Recv:**

```
MPI_Ssend(local_data, N, MPI_DOUBLE, up, ...);
MPI_Recv(upper_halo, N, MPI_DOUBLE, up, ...);
```

bloccante ↠ *il processo prima manda, poi riceve*

2. **Isend / Irecv:**

```
MPI_Ssend(local_data, N, MPI_DOUBLE, up, ...);
MPI_Recv(upper_halo, N, MPI_DOUBLE, up, ...);
```

non bloccante ↠ *il processo continua a fare calcoli*

3. **Sendrecv;**

```
MPI_Sendrecv(local_data, N, MPI_DOUBLE, up, ...,
  upper_halo, N, MPI_DOUBLE, up, ...);
```

bloccante ↠ *Funzione combinata che fa invio e ricezione simultanea in un'unica chiamata:*

APPROCCIO 2 - SCAMBIO DI HALO TRA PROCESSI

Procedimento

1. Il processo Master (rank = 0) distuibuisce le righe, sendcounts, senddispls

```
MPI_Scatterv(
    A,
    sendcounts,
    displs_scatterv,
    MPI_DOUBLE,
    recvbuf,
    sendcounts[0],
    MPI_DOUBLE,
    0,
    MPI_COMM_WORLD
);
```

```
// Broadcast sendcounts and senddispls to all processes
MPI_Bcast(sendcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(senddispls, size, MPI_INT, 0, MPI_COMM_WORLD);
```

APPROCCIO 2 - SCAMBIO DI HALO TRA PROCESSI

Procedimento

2. Ogni processo prepara i vicini e i buffer:
   - calcolo dei vicini:

```
int up = (my_rank > 0) ? my_rank - 1 : MPI_PROC_NULL
```

```
int down = (my_rank < size - 1) ? my_rank + 1 : MPI_PROC_NULL
```

   - Calcolo del buffer con halo:

```
int total_rows = my_rows + 2
```

```
double * my_A_plus_halos = calloc(total_rows * N, sizeof(double))
```

   - Puntatori:

```
double *upper_halo = my_A_plus_halos
```

```
double *local_data = my_A_plus_halos + N
```

```
double *lower_halo = my_A_plus_halos + (my_rows + 1) * N
```

   ➳ Permettono di riferirsi facilmente agli halo e ai dati locali.

   - Copia dei dati locali nel buffer centrale:

```
memcpy(local_data, my_A, my_rows * N * sizeof(double));
```

APPROCCIO 2 - SCAMBIO DI HALO TRA PROCESSI

Procedimento

2. Scambio degli halo:

Exchange mode 0: SSend / Recv (bloccante) ↠ *Semplice da capire, ma rischio deadlock se l'ordine non è corretto*

- Bloccante ↠ invio/ricezione sequenziali.
- Prima invio verso l'alto, poi ricezione dell'halo superiore.
- Poi gestione dell'halo inferiore: prima ricezione, poi invio.

Exchange mode 1: Isend / Irecv (non bloccante)

- Si fanno MPI_Irecv per ricevere gli halo dai vicini.
- Poi MPI_Isend per inviare le proprie righe ai vicini.
- Alla fine MPI_Waitall attende il completamento di tutte le comunicazioni.

Exchange mode 2: Sendrecv (bloccante combinato)

- Si fa MPI_Sendrecv: invio e ricezione in un'unica chiamata.
- Si scambiano simultaneamente riga superiore e riga inferiore con i vicini.

APPROCCIO 2 - SCAMBIO DI HALO TRA PROCESSI

Procedimento

3. Calcolo locale e invio risultati

```
// Computer local sum
for (int i = 0; i < my_rows; i++) {
    for (int j = 0; j < N; j++){ // for each column

        // initialize accumulators
        double sum = 0; // sum of the neighborhood values
        int count = 0;  // number of elements, for the mean

        // Loop over the 3x3 neighborhood
        for (int di = -1; di <= 1; di++) {
            int ni = i + di + 1; // +1 to skip the upper halo

            // check MPI halo boundaries
            if ((i == 0 && my_rank == 0 && di < 0) ||
                (i == my_rows - 1 && my_rank == size - 1 && di > 0)) {
                continue;
            }

            for (int dj = -1; dj <= 1; dj++) {
                int nj = j + dj;

                // horizontal boundaries
                if (nj < 0 || nj >= N) continue;

                sum += my_A_plus_halos[ni * N + nj];
                count++;
            }
        }


        my_T[i * N + j] = (local_data[i * N + j] * count > sum) ? 1 : 0; // ottimizzazione
    }
}
```

```
MPI_Gatherv(
    my_T,
    my_rows * N,
    MPI_INT,
    T,
    sendcounts,
    senddispls,
    MPI_INT,
    0,
    MPI_COMM_WORLD
);
```

STRONG SCALING

I tempi mostrati corrispondono alla media di **2 rilevamenti indipendenti**

Lo **speedup teorico** è determinato mediante la formula: $S_p = \frac{T_1}{T_p}$

L'**efficiency** è determinato mediante la formula: $E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p}$

P	N	Mean Time [s]	Speedup	Efficiency
1	5000	1.286197	1.00	1.00
2	5000	0.678363	1.90	0.95
4	5000	0.389227	3.30	0.83
8	5000	0.256494	5.01	0.63
12	5000	0.221677	5.80	0.48
16	5000	0.208981	6.15	0.38
20	5000	0.212893	6.04	0.30
24	5000	0.208343	6.17	0.26
48	5000	0.291341	4.41	0.09

Performance using M-S Block Distribution

P	N	Mean Time [s]	Speedup	Efficiency
1	5000	1.091230	1.00	1.00
2	5000	0.579273	1.88	0.94
4	5000	0.331944	3.29	0.82
8	5000	0.223750	4.88	0.61
12	5000	0.190272	5.74	0.48
16	5000	0.166460	6.55	0.41
20	5000	0.161925	6.74	0.34
24	5000	0.159344	6.85	0.29
48	5000	0.185362	5.89	0.12

Performance using Ssend/Recv

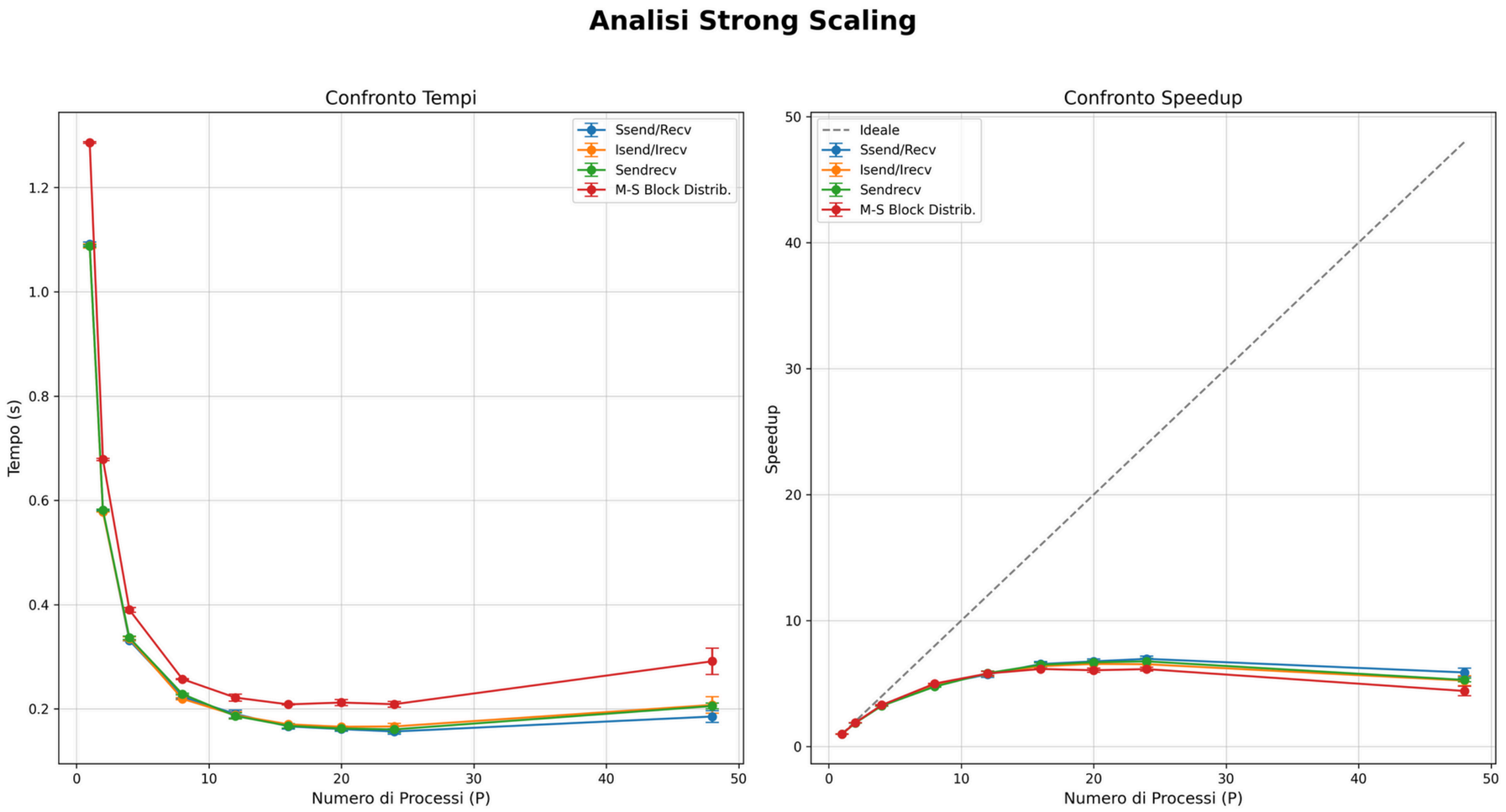
P	N	Mean Time [s]	Speedup	Efficiency
1	5000	1.087288	1.00	1.00
2	5000	0.577604	1.88	0.94
4	5000	0.334618	3.25	0.81
8	5000	0.219390	4.95	0.62
12	5000	0.187958	5.78	0.48
16	5000	0.170416	6.38	0.40
20	5000	0.165077	6.58	0.33
24	5000	0.166345	6.54	0.27
48	5000	0.207652	5.24	0.11

Performance using Isend/Recv

P	N	Mean Time [s]	Speedup	Efficiency
1	5000	1.087809	1.00	1.00
2	5000	0.581190	1.87	0.94
4	5000	0.336371	3.23	0.81
8	5000	0.228509	4.76	0.60
12	5000	0.186738	5.82	0.49
16	5000	0.167508	6.49	0.41
20	5000	0.162525	6.69	0.33
24	5000	0.160025	6.80	0.28
48	5000	0.205559	5.29	0.11

Performance using Sendrecv

STRONG SCALING



- La **dimensione** della matrice rimane **costante** mentre **aumenta il numero di thread o nodi**.
- Il **lavoro totale** da eseguire rimane **invariato**.
- Il **lavoro** assegnato a ciascun **nodo**/**thread diminuisce al crescere del numero di nodi**.
- La **legge di Amdahl** fornisce un **limite teorico** allo speedup ottenibile a causa della frazione seriale del codice.

$$S_p = \frac{1}{(1-f) + \frac{f}{p}}$$

dove:

- f è la frazione del codice parallelizzabile,
- $(1-f)$ è la frazione seriale,
- p è il numero di processori/thread.

WEAK SCALING

I tempi mostrati corrispondono alla media di **2 rilevamenti indipendenti**

Lo **speedup teorico** è determinato mediante la formula:

$$S_p^{\text{scaled}} = P \cdot \frac{T_1}{T_p}$$

L'**efficiency** è determinato mediante la formula:

$$E_p^{\text{scaled}} = \frac{S_p^{\text{scaled}}}{P} = \frac{T_1}{T_p}$$

P	N	Mean Time [s]	Speedup	Efficiency
1	2000	0.208718	1.00	1.00
2	2828	0.218468	1.91	0.95
4	4000	0.252308	3.31	0.83
8	5656	0.334556	5.00	0.62
12	6924	0.411649	6.08	0.51
16	8000	0.499857	6.68	0.42
20	8940	0.621941	6.71	0.34
24	9792	0.743944	6.74	0.28
48	13872	1.741691	5.75	0.12

M-S Block Distribution

P	N	Mean Time [s]	Speedup	Efficiency
1	2000	0.174929	1.00	1.00
2	2828	0.186428	1.88	0.94
4	4000	0.213681	3.28	0.82
8	5656	0.276021	5.07	0.63
12	6924	0.335132	6.26	0.52
16	8000	0.416826	6.71	0.42
20	8940	0.469409	7.45	0.37
24	9792	0.518723	8.10	0.34
48	13872	0.909133	9.24	0.19

Ssend/Recv

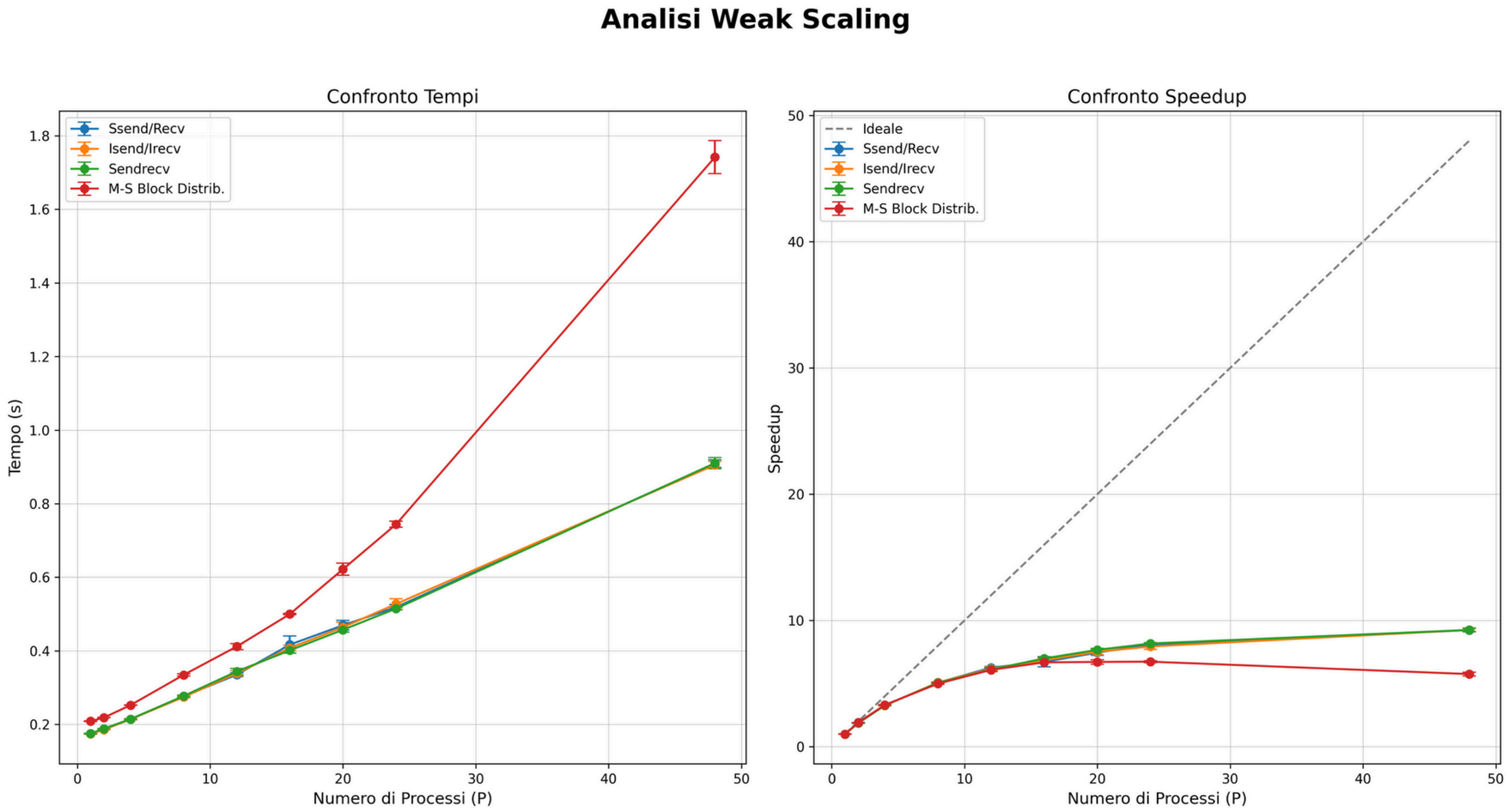
P	N	Mean Time [s]	Speedup	Efficiency
1	2000	0.174469	1.00	1.00
2	2828	0.185591	1.88	0.94
4	4000	0.214037	3.26	0.81
8	5656	0.274307	5.09	0.64
12	6924	0.338743	6.18	0.52
16	8000	0.407815	6.84	0.43
20	8940	0.464168	7.51	0.38
24	9792	0.527450	7.94	0.33
48	13872	0.905444	9.25	0.19

Isend/recv

P	N	Mean Time [s]	Speedup	Efficiency
1	2000	0.175303	1.00	1.00
2	2828	0.188511	1.86	0.93
4	4000	0.214424	3.27	0.82
8	5656	0.277186	5.06	0.63
12	6924	0.343902	6.12	0.51
16	8000	0.401306	6.99	0.44
20	8940	0.457216	7.67	0.38
24	9792	0.514770	8.17	0.34
48	13872	0.910038	9.24	0.19

Sendrecv

WEAK SCALING



- Si mantiene **costante il carico di lavoro per singolo nodo/thread.**
- La **dimensione totale** del problema **aumenta proporzionalmente** al numero di nodi.
- Questo scenario studia la scalabilità debole dell'algoritmo.
- La **legge di Gustafson** permette di prevedere lo **speedup teorico** quando il problema cresce con il numero di processori.

$$S_p = p - (1 - f) \cdot (p - 1)$$

dove:

- f è la frazione del codice parallelizzabile,
- $(1 - f)$ è la frazione seriale,
- p è il numero di processori/thread.

CONCLUSIONI

CONCLUSIONI

Considerazioni finali

- **OpenMP** ha mostrato ottimi risultati in termini di speedup ed efficiency, grazie al basso overhead di sincronizzazione. Si osserva inoltre che, con un **numero elevato di thread** e valori limitati nella matrice, le performance beneficiano di una **migliore località di cache**, che **riduce i cache miss** e migliora l'efficienza del calcolo parallelo, sfruttando al meglio la memoria veloce disponibile sui core.
- **MPI** risulta **meno efficiente** a causa del costo di comunicazione e della gestione degli halo.

MPI – confronto approcci

- **Approccio 1:** richiede un tempo di esecuzione maggiore e fornisce prestazioni peggiori, principalmente a causa del master thread che invia dati a tutti gli altri thread, che diventa un **collo di bottiglia** nel modello di comunicazione.
- **Approccio 2**: l'uso di MPI_Sendrecv fornisce i risultati migliori.

↠ **Complessivamente il problema si adatta meglio a un paradigma a memoria condivisa come OpenMP, mentre l'approccio MPI risente fortemente dell'overhead di comunicazione, che ne limita scalabilità ed efficienza.**