

Relazione sull'esercitazione di

Calcolo parallelo

Utilizzo delle librerie MPI

Bellicano Fabrizio

Marsano Matilde

Corso di Ingegneria del Software - Anno Accademico
2010/2011

Esercizio 1

Obiettivi dell'esercitazione

Questo esercizio è una introduzione sull'uso di MPI (Message Passing Interfaces), una libreria che implementa IPC (Inter-Process Communications) tra processi paralleli che girano all'interno di un cluster.

Contenuto del 1° esercizio

Richiesta 1

Eseguire il codice sul cluster IMATI, descrivere, in maniera sintetica, la struttura ed il comportamento atteso e quello visibile eseguendo il codice. In questa descrizione porre particolare attenzione all'individuazione della modalità di interazione tra processi e al codice MPI presente nel programma.

Richiesta 2

Modificare il codice in modo tale che sia il processo con identificatore più alto a funzionare da console e l'anello dei processi sia percorso dal processo con identificatore più alto al processo 0 e ripetere il passo 1 con questa versione.

Svolgimento

Prima richiesta

Segue codice sorgente utilizzato:

```

1  /*****
2   * Questo programma e' stato sviluppato da:
3   * Open Systems Laboratory
4   * http://www.lam-mpi.org/tutorials/
5   * Indiana University
6   *
7   * Gli n processi comunicano su un anello:
8   * il processo 0 iniva al processo 1 e riceve da n-1;
9   * 1 riceve da 0 e invia a 2;
10  * ...
11  * n-1 riceve da n-2 e invia a 0;
12  *
13  * Il codice seguente fornisce la struttura generale
    del
14  * programma e' necessario completarlo con le opportune
    primitive
15  * MPI per lo scambio di messaggi (MPI_Send, MPI_Recv)
16  */
17
18 #include <stdio.h>
19 #include <mpi.h>
20
21
22 int main(int argc, char *argv[])
23 {
24     MPI_Status status;
25     int num, rank, size, tag, next, from;
26
27     /* Start up MPI */
28     MPI_Init(&argc, &argv);
29
30
31     MPI_Comm_rank(MPLCOMM_WORLD, &rank);
32
33     /* Determina la dimensione del gruppo di processori
        */
34     MPI_Comm_size(MPLCOMM_WORLD, &size);
35

```

```

36
37
38     tag = 201;
39     next = (rank + 1) % size;
40     from = (rank + size - 1) % size;
41
42     /* Se siamo il processo "console", ossia quello
43     con rank nullo, [vuol dire che sta partendo un
44     nuovo giro!]
45     chiediamo all'utente di inserire un numero intero,
46     per specificare quante volte vogliamo "girare"
47     nell'anello
48     */
49     if (rank == 0)
50     {
51         printf("Enter the number of times around the ring: \n");
52         scanf("%d", &num);
53
54         printf("Process %d sending %d to %d\n", rank, num,
55             next);
56
57         /* Invia il numero al prossimo processo nel ring:
58            - MPI_Send(&buf, count, MPI_Datatype, dest, tag
59            , MPLCOMM_WORLD);
60            - MPI_Recv(&buf, count, MPI_Datatype, source,
61            tag, MPLCOMM_WORLD, &status);
62         La sintassi di MPI_Send() e MPI_Recv() la
63         seguente:
64         buf punta al primo elemnto del dato che si
65         vuole inviare, oppure, qualora il dato debba
66         essere ricevuto la destinazione.
67         count il numero di elementi di tipo
68         MPI_Datatype da trasferire alla destinazione
69         dest dalla sorgente source.
70         status contiene informazioni relative al
71         messaggio.

```

```

62
63     */
64     MPI_Send(&num, 1, MPI_INT, next, tag,
              MPLCOMM_WORLD);
65 }
66
67
68 do {
69     /* Riceve il numero,
70        se il processo console non ha iniziato ad
71        inviare, gli altri si fermano in attesa del
72        dato. */
73
74     MPI_Recv(&num, 1, MPI_INT, from, tag,
              MPLCOMM_WORLD, &status);
75
76     printf("Process %d received %d\n", rank, num);
77
78     if (rank == 0)
79     {
80         --num;
81         printf("Process 0 decremented %d\n", num);
82     }
83
84     printf("Process %d sending %d to %d\n", rank, num,
            next);
85
86     /* Invia il numero al prossimo processo del ring */
87     MPI_Send(&num, 1, MPI_INT, next, tag,
              MPLCOMM_WORLD);
88 } while (num > 0);
89
90 printf("Process %d exiting\n", rank);
91
92 /* L'ultimo processo effettua un invio ulteriore al
93    processo 0, che si pone in attesa di questo prima
94    di poter uscire */

```

```

92  if (rank == 0)
93      MPI_Recv(&num, 1, MPI_INT, from, tag,
               MPI_COMM_WORLD, &status);
94
95  /* Quit */
96      MPI_Finalize();
97
98  return 0;
99  }

```

Note sulla prima richiesta

L'esercizio richiede inizialmente di eseguire il codice sul cluster IMATI. Il programma permette a n processi di comunicare tra loro simulando una struttura ad anello: un messaggio, quindi, verrà inviato un numero di volte prestabilito a ciascun processo. L'intero codice é eseguito da ciascun processore che, appena entrato nel *main* inizializza l'ambiente MPI e identifica il proprio *rank* all'interno di MPI_COMM_WORLD attraverso la funzione *MPI_Comm_rank()*. Successivamente, dopo aver ottenuto anche il numero totale di processi attivi (*MPI_Comm_size()*), esegue la sua porzione di codice. Il processo con *rank* = 1, che chiameremo console, ha il compito di interrogare l'utente e ottenere il numero di giri da effettuare attorno all'anello, cioè il numero di volte che ciascun processo invierà e riceverà il messaggio. Mentre il processo console esegue le inizializzazioni, i rimanenti *size-1* lavoratori restano bloccati in attesa di ricevere un messaggio (*MPI_Recv()*). La console allora invierà il messaggio al processo con *rank* = 1, il quale lo invierà al processo successivo (*rank* = 2) e così via in ordine crescente; l'ultimo processo (*rank* = $n-1$) ha il compito di far tornare il messaggio alla console, che si occuperà di decrementare il numero di giri attorno all'anello ancora da effettuare.

Il lavoro termina con la ricezione dell'ultimo messaggio da parte della console.

Seconda richiesta

Segue codice sorgente utilizzato:

```
1  /*****
2   * Questo programma e' stato sviluppato da:
3   * Open Systems Laboratory
4   * http://www.lam-mpi.org/tutorials/
5   * Indiana University
6   *
7   * Gli n processi comunicano su un anello:
8   * il processo 0 invia al processo 1 e riceve da n-1;
9   * 1 riceve da 0 e invia a 2;
10  * ...
11  * n-1 riceve da n-2 e invia a 0;
12  *
13  * Il codice seguente fornisce la struttura generale
    del
14  * programma e' necessario completarlo con le opportune
    primitive
15  * MPI per lo scambio di messaggi (MPI_Send, MPI_Recv)
16  */
17
18 #include <stdio.h>
19 #include <mpi.h>
20
21
22 int main(int argc, char *argv[])
23 {
24     MPI_Status status;
25     int num, rank, size, tag, next, from;
26
27     /* Start up MPI */
28     MPI_Init(&argc, &argv);
29
30     MPI_Comm_rank(MPLCOMM_WORLD, &rank);
31
32     /* Determina la dimensione del gruppo di processori
        */
```

```

33 MPI_Comm_size(MPLCOMM_WORLD, &size);
34
35
36
37 tag = 201;
38 num = 1;
39 from = (rank + 1) % size;
40 next = (rank + size - 1) % size;
41
42
43 /* Se siamo il processo "console", quello con rank
      nullo, [vuol dire sta partendo un nuovo giro!]
   chiediamo all'utente di inserire un numero intero,
   per specificare quante volte vogliamo "girare"
   all'interno dell'anello
44 */
45
46 /* Purtroppo solo il processo 0 pu leggere su scanf
   , ricevo da zero e per prima cosa lo invio a
   console! */
47
48 if (rank == 0)
49 {
50     printf("Enter the number of times around the ring: \n");
51     scanf("%d", &num);
52
53     MPI_Send(&num, 1, MPI_INT, (size - 1), tag,
54             MPLCOMM_WORLD);
55 }
56 if (rank == (size - 1))
57 {
58     MPI_Recv(&num, 1, MPI_INT, 0, tag, MPLCOMM_WORLD,
59             &status);
60     printf("START Process %d sending %d to %d\n", rank
61           , num, next);

```



```

60     MPI_Send(&num, 1, MPI_INT, next, tag,
61             MPLCOMM_WORLD);
62 }
63
64
65 do {
66     /* Riceve il numero,
67        se il processo console non ha iniziato ad
68        inviare, gli altri si fermano in attesa del
69        dato. */
70
71     MPI_Recv(&num, 1, MPI_INT, from, tag,
72             MPLCOMM_WORLD, &status);
73
74     printf("Process %d received %d\n", rank, num);
75
76     if (rank == (size - 1))
77     {
78         --num;
79         printf("Process %d decremented %d\n", rank);
80     }
81
82     printf("Process %d sending %d to %d\n", rank, num,
83           next);
84
85     /* Invia il numero al prossimo processo del ring */
86     MPI_Send(&num, 1, MPI_INT, next, tag,
87             MPLCOMM_WORLD);
88 } while (num > 0);
89
90 printf("Process %d exiting\n", rank);
91
92 /* L'ultimo processo effettua un invio ulteriore al
93    processo 0, che si pone in attesa di questo prima
94    di poter uscire */
95
96 if (rank == (size - 1))

```

```

90      MPI_Recv(&num, 1, MPI_INT, from, tag,
               MPLCOMM_WORLD, &status);
91
92      /* Quit */
93      MPI_Finalize();
94
95      return 0;
96  }

```

Note sulla seconda richiesta

La seconda richiesta indicava di fare scorrere il messaggio nell'anello in senso contrario, ossia dal processo $n-1$ al processo 0 .

La soluzione più intuitiva, ossia quella di modificare i ruoli definiti alla riga di codice 48:

```
if ( rank == (size - 1) )
```

non è stata possibile, in quanto l'attuale versione di MPI consente **solo al processo master** di leggere da *stdin* (funzioni *scanf*, *getc*, *sscanf*, *gets*).

Ci sono varie soluzioni per ovviare al problema. La migliore sarebbe far leggere da file, al processo con $rank = n-1$, il numero di giri da effettuare sul ring. Leggendo tramite *fscanf* (non è *standard input*), il nuovo programma console può acquisire il dato nascondendo l'inconveniente. Abbiamo, tuttavia, scartato questa soluzione a causa della poca "interattività" che il metodo consentirebbe; inoltre un approccio del genere, implicherebbe accessi a disco, che su un'architettura distribuita potrebbero avere ripercussioni negative, quali rallentamento del runtime e tempi lunghi nel seek del file. Questa ipotesi è stata fatta dopo aver lanciato il comando *cat /etc/fstab*, che ha mostrato un filesystem **non** distribuito sul cluster, ma un normale ext3. L'approccio scelto trascura nel dettaglio requisiti, ma ottiene i risultati richiesti. Abbiamo, infatti lasciato invariata l'acquisizione del messaggio da parte del processo con $rank = 0$, tramite lettura *stdin*, invertito i processi definiti come *next* e *from* e inviato tramite *MPI_Recv()* il numero al processo $n-1$, prima di iniziare la vera e propria comunicazione sull'anello.

Il resto del programma, quindi, rimane invariato, e l'anello verrà percorso in senso opposto al precedente, ovvero dal rank maggiore a quello nullo.

Esercizio 2

Scopo dell'esercitazione

Il programma implementa la struttura di un farm in C-MPI con attribuzione dinamica del carico. Il dominio dei dati in input deve essere partizionato in modo da generare un numero di sottodomini indipendenti (task) che sia maggiore (ad esempio multiplo) rispetto al numero dei processi paralleli (worker) utilizzati.

Contenuto del 2° esercizio

Analizziamo come, grazie all'impiego di più processori su un'architettura distribuita, il tempo di esecuzione di un algoritmo che svolge operazioni aritmetiche viene notevolmente ridotto suddividendo dinamicamente il carico di lavoro.

Svolgimento

Segue codice sorgente utilizzato:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define WORKTAG 1
#define DIETAG 2
#define CHUNKTAG 3
#define ARRAYSIZE 160000
```

```

float data[ARRAYSIZE];

    /* Local functions */

static void master(void);
static void slave(void);
static void get_next_work_item(int, int*);
static float do_work(int);


int main(int argc, char **argv)
{
    int myrank;

    /* Inizializzazione MPI */
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPLCOMM_WORLD, &myrank);

    if (myrank == 0)
        master();
    else
        slave();

    /* Finalizzo MPI */
    MPI_Finalize();

    return 0;
}


static void master(void)
{

```

```

    int ntasks, work;
    float result;
    int chunksize, dest, n;

    double starttime, endtime;
    MPI_Status status;

    /* Acquisizione del numero di processi */
    MPI_Comm_size(MPLCOMM_WORLD, &ntasks);

    /* Inizializzo array, inserisco dati casuali */
    int rank;
    for(rank = 0; rank < ARRAYSIZE; ++rank)
        data[rank] = rank * 1.0;

    printf("Dimensione Array: %d\nInserire numero di parti in cui si vuole suddividere il lavoro:\n", ARRAYSIZE);
    scanf("%d",&n);
    if(!(ARRAYSIZE % n == 0)){
        printf("Errore, il coefficiente inserito non divide l'array in parti intere.\n");
        exit(1);
    }

    /* Misuro il tempo di esecuzione del lavoro */
    starttime = MPI_Wtime();

    chunksize = (ARRAYSIZE / n);

    /* Ottenimento primo lavoro */
    work = 0;
    get_next_work_item(0,&work);

    /* Assegnamento del lavoro a ciascuno slave (rank da 1 a ntasks-1). Ogni schiavo riceve la dimensione del lavoro e la porzione di array su cui dovr

```

```

operare. */
    for (dest=1; dest<ntasks; ++dest)
    {
        MPI_Send(&chunksize, 1, MPI_INT, dest,
            CHUNKTAG, MPLCOMM_WORLD);
        MPI_Send(&data[work], chunksize,
            MPI_FLOAT, dest, WORKTAG,
            MPLCOMM_WORLD);

        get_next_work_item(chunksize,&work);
    }

    float avrg = 0;

    while (work != -1) {

        MPI_Recv(&result, 1, MPI_FLOAT,
            MPL_ANY_SOURCE, MPL_ANY_TAG,
            MPLCOMM_WORLD, &status);

        avrg += result;

        /* Invio lavoro successivo: */
        MPI_Send(&chunksize, 1, MPI_INT, status
            .MPL_SOURCE, CHUNKTAG,
            MPLCOMM_WORLD);
        MPI_Send(&data[work], chunksize,
            MPI_FLOAT, status.MPL_SOURCE,
            WORKTAG, MPLCOMM_WORLD);

        get_next_work_item(chunksize,&work);
    }

    /* Finiti i lavori il master riceve tutti gli
       ultimi risultati in lavorazione */

    for (rank = 1; rank < ntasks; ++rank)

```

```

    {
        MPI_Recv(&result , 1, MPI_FLOAT,
                 MPL_ANY_SOURCE, 0, MPL_COMM_WORLD, &
                 status);
        avrg += result;
    }

    /* Il master infine ordina a ciascuno slave di
       terminare l'esecuzione tramite l'invio del tag
       DIETAG. */
    for (rank = 1; rank < ntasks; ++rank)
        MPI_Send(0, 0, MPI_INT, rank, DIETAG,
                 MPL_COMM_WORLD);

    avrg /= n;

    printf("La media degli elementi nell'array : \n
           %f\n", avrg);

    endtime = MPI_Wtime();
    printf("Tempo totale sul master: %f sec\n", (
        endtime - starttime));
}

static void slave(void)
{
    int chunk;
    float result;

    MPI_Status status;

    double starttimeS, endtimeS;
    starttimeS = MPI_Wtime();

    while (1)
    {

```

```

        MPI_Recv(&chunk, 1, MPI_INT, 0,
                 MPL_ANY_TAG, MPL_COMM_WORLD, &status
                 );

    /* Controlla il tipo di messaggio ricevuto tramite il
       tag, se il tag del messaggio
       ricevuto coincide con il tag che decreta il
       termine del lavoro esce dal programma. */
    if (status.MPI_TAG == DIETAG)
    {
        endtimeS = MPI_Wtime();
        printf("Tempo di esecuzione sul
               worker: %f sec\n", (endtimeS
                                   - starttimeS));
        return;
    }

    MPI_Recv(&data[0], chunk, MPI_FLOAT, 0,
             WORKTAG, MPL_COMM_WORLD, &status);

    /* Chiamata alla funzione che esegue il lavoro */
    result = do_work(chunk);

    /* Invio del risultato parziale al master (rank = 0)
       */
    MPI_Send(&result, 1, MPI_FLOAT, 0, 0,
             MPL_COMM_WORLD);

}

}

/* La funzione get_next_work_item riceve come argomenti
   la dimensione del lavoro e il riferimento alla
   variabile che tiene conto dell'ultimo lavoro
   assegnato dal master. count viene aggiornata dalla
   funzione e, modificando il puntatore si modifica

```



```

    anche la variabile puntata da esso che appartiene al
    main: cosi viene ottenuto il nuovo lavoro. Nel caso
    i lavori fossero esauriti count assume un valore
    convenzionale. */
static void get_next_work_item(int chunk, int *count)
{
    if(chunk == 0)
    {
        *count = 0;
        return;
    }
    else{
        *count += chunk;
        /* Controllo di avere ancora lavori disponibili */
        if(*count > (ARRAYSIZE-1))
            *count = -1;
        return;
    }
}

/* Il programma calcola la media degli elementi nell'
   array data. */
static float do_work(int chunk)
{
    int i;
    float avg = 0.0;

    for(i=0; i < ARRAYSIZE; i++)
        avg += data[i];
    avg /= chunk;
    return avg;
}

```

Note sul codice

Descriviamo brevemente come funziona il programma. Vengono impiegati n processi paralleli; di questi, quello con $rank = 0$ viene chiamato master e i restanti $n-1$ processi slave. Il master inizia per primo a lavorare: inizializza i dati, chiede all'utente il numero di sottodomini da creare ed invia ad ogni slave il primo task. L'invio di ciascun sottodominio avviene tramite due funzioni *MPI_Send()*. La prima contiene informazioni circa la dimensione del sottodominio assegnato, la seconda invece invia proprio la porzione di array su cui lo slave dovrà operare. Avendo infatti dichiarato come variabile globale l'array `data[]`, ogni processo ne possiede uno; ciascuno slave non fa altro che ricevere parte del vettore appartenente al master per copiarlo all'interno del proprio (inizialmente vuoto).

Assegnata a ciascuno slave la porzione di array, il master entra in un loop nel quale attende un risultato da uno slave qualunque (*MPI_ANY_SOURCE*). Allo slave che invia il dato (*status.MPI_SOURCE*) il master manda immediatamente il lavoro successivo (selezionato attraverso la funzione *get_next_job_item()*). Al termine dei tasks viene inviato l'avviso di uscita. Ciascuno slave può ricevere dal master tre tipi di messaggio: la dimensione dell'array su cui dovrà lavorare (identificato dal tag *CHUNKTAG*), la porzione di array da lavorare (identificato dal tag *WORKTAG*) oppure il messaggio di fine processo (identificato dal tag *DIETAG*). I primi due messaggi viaggiano in coppia, l'ultimo invece è un messaggio singolo. Lo slave si mette in attesa del primo messaggio, controlla che non sia l'avviso di uscita, qualora non lo fosse si mette nuovamente in attesa per ricevere i dati. Grazie ai tag i processi non rischiano di confondere i messaggi ricevuti.

Analisi delle prestazioni

Come richiesto dalle consegne abbiamo aggiunto al codice le primitive *MPI_Wtime()* per raccogliere informazione sul tempo totale impiegato dal master e quello impiegato da ciascun worker. I risultati ottenuti compilando e facendo eseguire il codice sono riportati in *Figura 1*, *Figura 2* e sono concordi con quanto ci aspettavamo di osservare. Variando il numero di processori e dimensione dell'input, i risultati computazionali variano. Ai fini del test, abbiamo mantenuto costante la dimensione del dominio dei dati (160000).

```
mafalda@mafalda-laptop: ~
gruppo811@paperoga:~/Esercitazione2011

[gruppo811@paperoga Esercitazione2011]$ mpirun -np 3 ./farm
Dimensione Array: 160000
Inserire numero di parti in cui si vuole suddividere il lavoro:
400
La media degli elementi nell'array è: 79999.531250
Tempo di esecuzione sul worker: 22.230443 sec
Tempo di esecuzione sul worker: 22.231956 sec
Tempo totale sul master: 20.401061 sec
[gruppo811@paperoga Esercitazione2011]$
```

Figura 1: Utilizzo di 3 processori su 400 sottodomini.

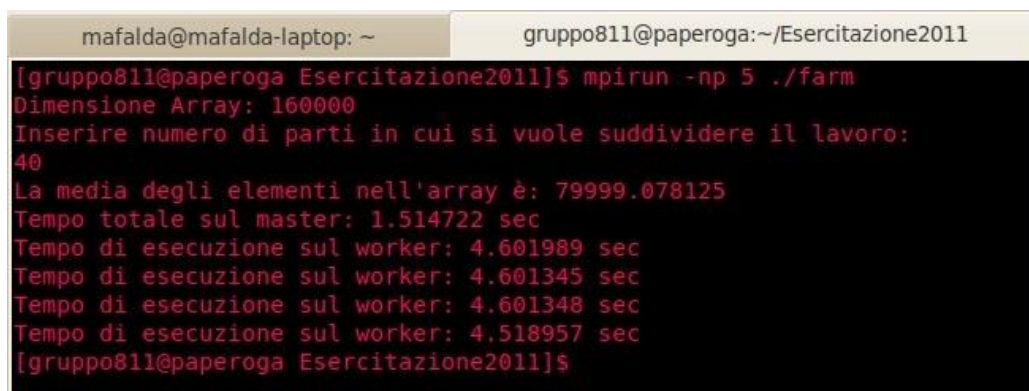
```
mafalda@mafalda-laptop: ~
gruppo811@paperoga:~/Esercitazione2011

[gruppo811@paperoga Esercitazione2011]$ mpirun -np 5 ./farm
Dimensione Array: 160000
Inserire numero di parti in cui si vuole suddividere il lavoro:
400
La media degli elementi nell'array è: 79999.531250
Tempo di esecuzione sul worker: 13.226691 sec
Tempo di esecuzione sul worker: 13.225363 sec
Tempo di esecuzione sul worker: 13.225376 sec
Tempo totale sul master: 10.571897 sec
Tempo di esecuzione sul worker: 13.026186 sec
[gruppo811@paperoga Esercitazione2011]$
```

Figura 2: Utilizzo di 5 processori su 400 sottodomini.

I risultati mostrano chiaramente come, mantenendo invariato dominio e sottodomini, i tempi di esecuzione dei task variano sensibilmente al variare del numero dei processori impiegati. In *Figura 1*, a causa dell'impiego di 3 processori, i tempi totali sono quasi raddoppiati rispetto a *Figura 2*.

A questo punto è utile analizzare come varia l'esecuzione del programma al variare del numero di sottodomini. Impieghiamo 5 processi paralleli (come in *Figura 2*), ma questa volta chiediamo al master di creare 40 sottodomini.



```
mafalda@mafalda-laptop: ~
gruppo811@paperoga:~/Esercitazione2011
[gruppo811@paperoga Esercitazione2011]$ mpirun -np 5 ./farm
Dimensione Array: 160000
Inserire numero di parti in cui si vuole suddividere il lavoro:
40
La media degli elementi nell'array è: 79999.078125
Tempo totale sul master: 1.514722 sec
Tempo di esecuzione sul worker: 4.601989 sec
Tempo di esecuzione sul worker: 4.601345 sec
Tempo di esecuzione sul worker: 4.601348 sec
Tempo di esecuzione sul worker: 4.518957 sec
[gruppo811@paperoga Esercitazione2011]$
```

Figura 3: Utilizzo di 5 processori su 40 sottodomini.

Al diminuire del numero di tasks i tempi di esecuzione diminuiscono anch'essi. Possiamo presupporre che questo sia dovuto, specie in questa esercitazione a scopo didattico, della notevole diminuzione del numero di primitive MPI *MPI_Send()* e *MPI_Recv()* effettuate da master e slaves per comunicare. Condizione limite è che il numero di lavori non sia minore del numero di processori; in tal caso non si apprezzerebbero le potenzialità della struttura a farm.

Per un'analisi più approfondita invece ci viene in aiuto il tool **jumpshot-4**, che ci consente di visualizzare su un diagramma di Gantt l'andamento dei processi e le loro comunicazioni, secondo la notazione:

Topo	Name ▼	V ▼	S ▼	count ▼	incl ▼	excl ▼
	Preview_Arrow	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	0	0	0
	message	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1202	2.567	0
	Preview_State	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	0	0	0
	MPI_Comm_rank	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	3	0	0
	MPI_Comm_size	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1	0	0
	MPI_Recv	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1202	2.836	2.836
	MPI_Send	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	1202	0.001	0.001
	Preview_Event	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	0	0	0
	MPE_Comm_finalize	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	3		0

Figura 4: Legenda del tool jumpshot

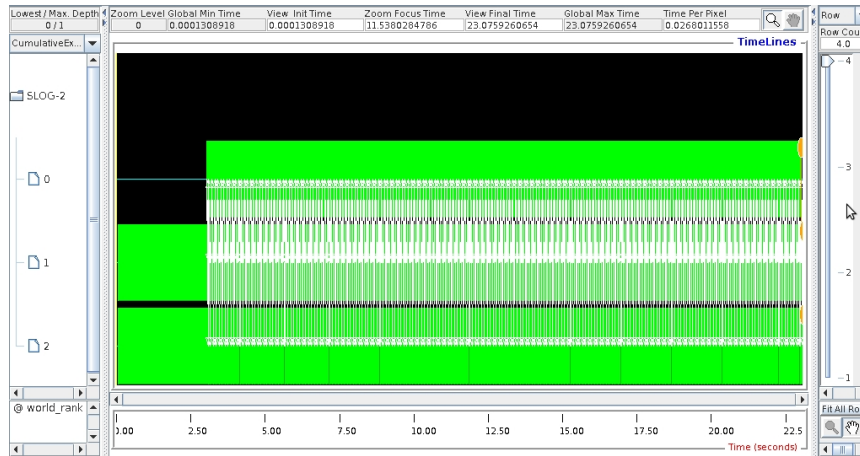


Figura 5: Jumpshot. Utilizzo di 3 processori su 400 sottodomini.

Queste immagini corrispondono all'esecuzione del codice riportato in *Figura 1*. Analizzando il grafico mettiamo in evidenza le seguenti cose: naturalmente i messaggi partono dal master e si susseguono molto ravvicinatamente dal punto di vista temporale. Essendo le comunicazioni molto fitte riportiamo anche uno zoom sull'immagine. (Figura 6)

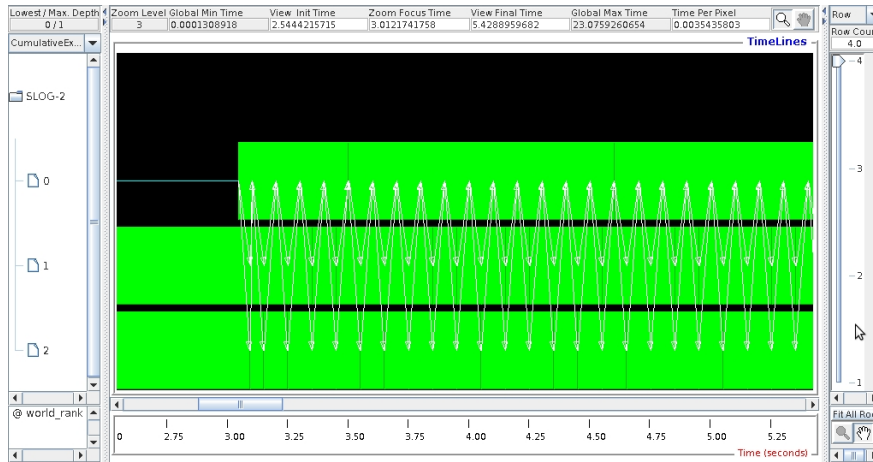


Figura 6: Jumpshot. Utilizzo di 3 processori su 400 sottodomini. (Zoom)

Grazie allo Zoom risulta evidente l'interazione tra i processi. Ogni volta che uno slave termina la computazione avvisa il master, che provvede ad inviare il nuovo *work*. Come evidenzia la legenda, l'istante in cui viene effettuata una *MPI_Send()* è colorato di blu (sebbene si veda poco), mentre gli intervalli temporali in cui i processi sono in attesa di *MPI_Recv()* sono verdi.

Vediamo ora cosa cambia nel diagramma utilizzando 5 processori:

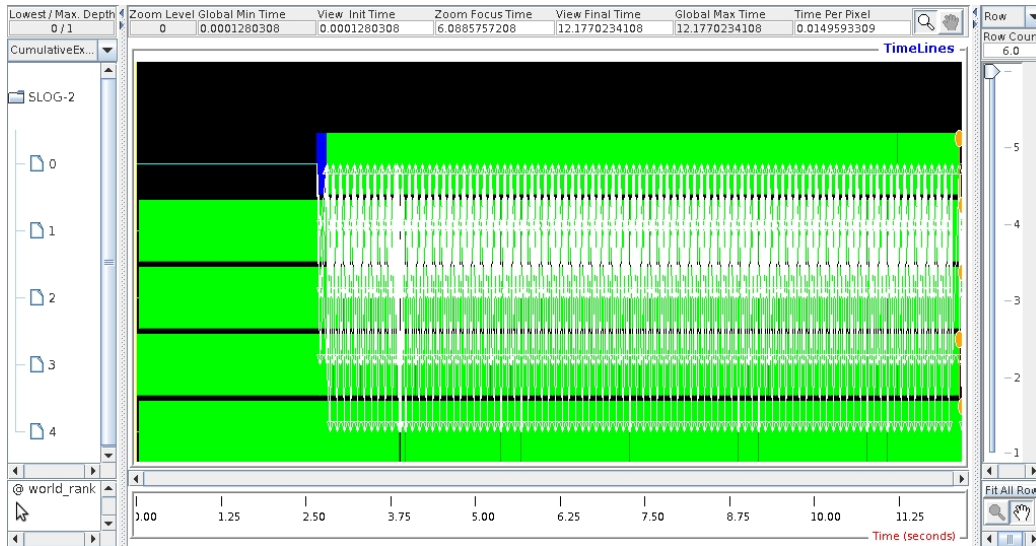


Figura 7: Jumpshot. Utilizzo di 5 processori su 400 sottodomini.

La differenza più evidente con le immagini precedenti è la netta diminuzione del tempo di esecuzione di master e slaves (tempo in ascissa). È interessante notare come all'avvio del processo master, l'intervallo di tempo dedicato alle *MPI_Send* (zone blu) occupi decisamente più spazio. Questo momento corrisponde nel codice al **for** di assegnazione dei primi lavori.

Riportiamo ora il risultato fornito da Jumpshot-4 a seguito dell'esecuzione del codice in *Figura 3*:

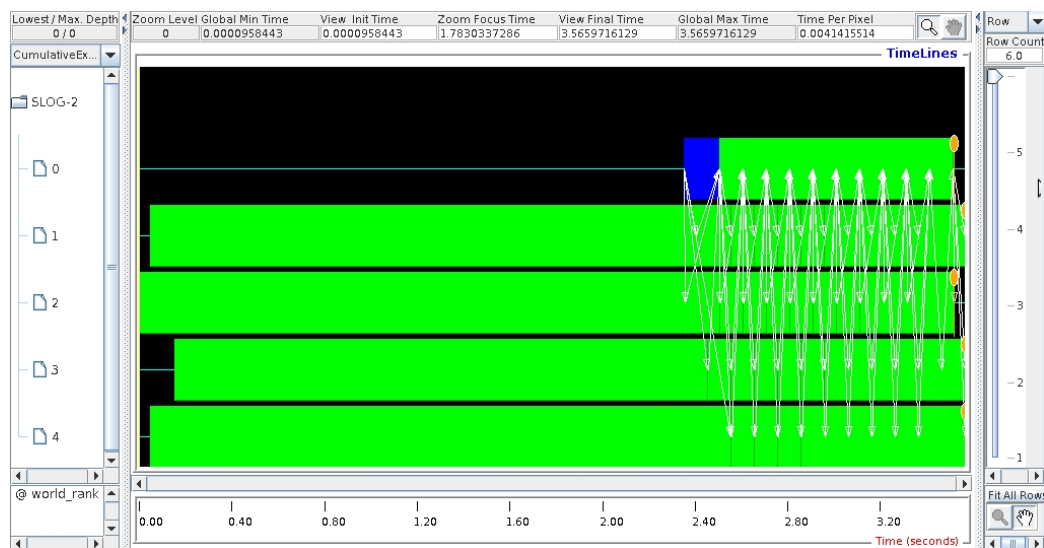


Figura 8: Jumpshot. Utilizzo di 5 processori su 40 sottodomini.

Conclusioni

Per nostra curiosità, abbiamo concluso l'esercitazione analizzando ancora due situazioni riportate in seguito.

In *Figura 9* abbiamo ipotizzato di non poter sfruttare i vantaggi del calcolo parallelo e abbiamo affidato, senza bilanciamento del carico, l'intero dominio ad un unico worker, mentre in *Figura 10* abbiamo diviso il dominio in tanti sottodomini quanto il numero di slaves. Questi sono i prevedibili risultati:

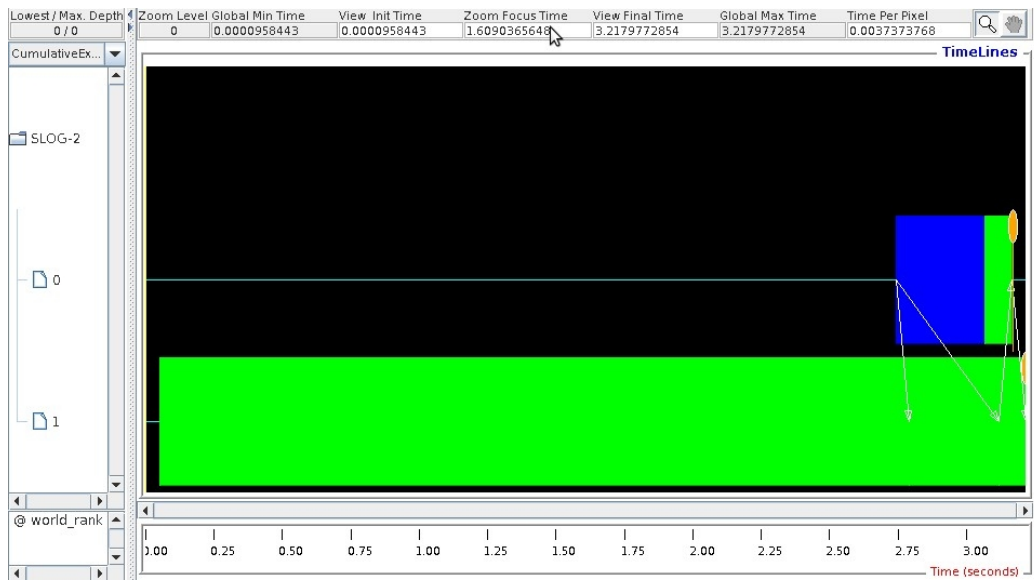


Figura 9: Jumpshot. Utilizzo di 2 processori senza partizionare il dominio.

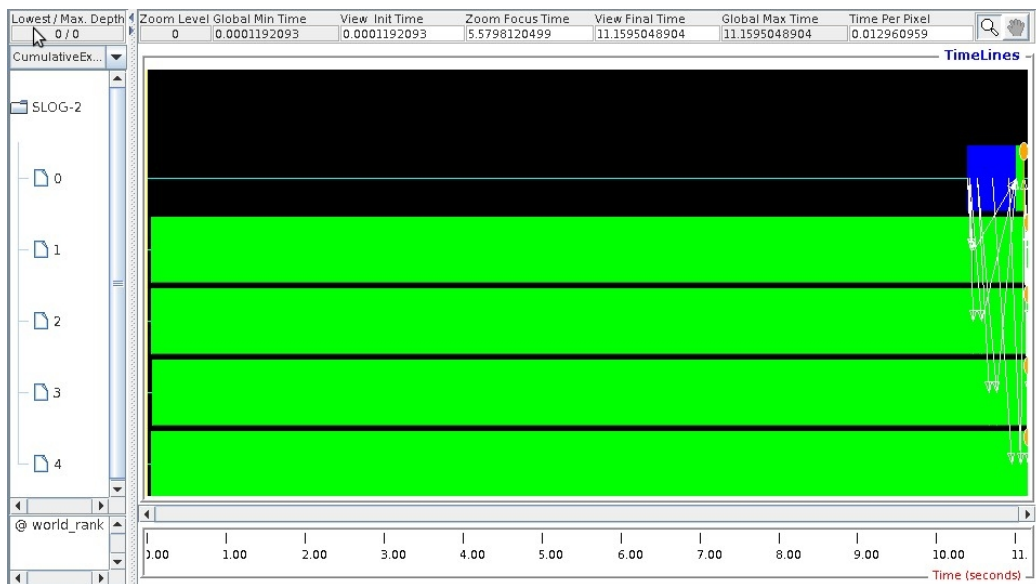


Figura 10: Jumpshot. Utilizzo di 5 processori su 4 sottodomini.