

Programmazione di Sistemi Multicore

Appunti del Corso

Alessandro Dori

Università degli Studi di Roma "La Sapienza"
Facoltà di Informatica
3° Anno, 1° Semestre



Anno Accademico 2024/2025

Contents

1	Introduzione	2
1.1	Why parallel computing	2
2	Memoria Distribuita: programmare con MPI	5
2.1	Sistemi a memoria distribuita	5
2.2	Single Program Multiple Data (SPMD)	5
2.3	MPI	5
2.4	Parallel Program Design	10
2.5	Parallel Design Patterns	10
2.6	Check for completion	12
2.7	Collective Communication	15
2.8	Performance Evaluation	23
2.9	Prodotti di matrici e vettori in MPI	31
2.10	MPI Derived Data Types	35
3	Pthreads	38
3.1	Matrix - Vector Multiplication in Pthreads	42
3.2	Sezioni critiche	43
3.3	Producer-consumer Synchronization and Semaphores	45
3.4	Barriers and Condition Variables	46
3.5	Read - Write Locks	49
3.6	Thread safety	54
3.7	Thread-safety in MPI	58
3.8	Caching	59
4	OpenMP	62

1 Introduzione

1.1 Why parallel computing

Dal 1986 al 2003, le prestazioni dei microprocessori sono aumentate, in media, del 50% all'anno (aumento di 60 volte in 10 anni). Dal 2003 questo aumento è rallentato. Dal 2015 al 2017, del 4% all'anno (aumento di 1,5 volte in 10 anni). Per questo motivo, invece di cercare di avere processori monolitici più potenti, stiamo mettendo più processori su un singolo circuito integrato.

Perché questo bisogno di prestazioni.

- L'umanità trova sempre problemi più complessi da affrontare:
 - Decodifica del genoma umano
 - Modellazione climatica
 - Ripiegamento delle proteine
 - Scoperta di farmaci
 - Ricerca energetica
 - Fisica fondamentale (ad esempio, esperimenti al CERN ecc...)

Tradizionalmente, l'aumento delle prestazioni è causato dall'aumento della densità dei transistor.

Nota

Ci sono limiti fisici che rendono necessari computer paralleli.

Perché costruiamo sistemi paralleli.

Alcuni costrutti di codifica possono essere riconosciuti e convertiti in un costrutto parallelo. Tuttavia, questo è spesso inefficiente. A volte, la soluzione migliore è fare un passo indietro e progettare un algoritmo completamente nuovo; cioè, spesso, non è sufficiente parallelizzare il codice sequenziale disponibile

Come scriviamo programmi paralleli?

- Parallelismo di attività
 - Partizione di vari compiti tra i core
 - Il “parallelismo temporale” che abbiamo visto nei circuiti è un tipo specifico di parallelismo di attività.
- Parallelismo dei dati
 - Partizione dei dati utilizzati per risolvere il problema tra i core

- Ogni core esegue operazioni simili sulla sua parte dei dati
- Simile al “parallelismo spaziale” nei circuiti

Se ogni core può funzionare in modo indipendente, è abbastanza simile a scrivere un programma seriale. Ad esempio, vuoi comprimere 100 file e hai 10 core: ogni core comprime 10 file. In pratica, i core devono coordinarsi, per motivi diversi:

- **Comunicazione:** ad esempio, un core invia la sua somma parziale a un altro core.
- **Bilanciamento del carico:** dividi il lavoro in modo uniforme in modo che uno non sia pesantemente caricato.
- **Sincronizzazione:** assicurati che i core finiscano il loro lavoro nello stesso momento.

Scrivremo programmi che sono esplicitamente paralleli utilizzando quattro diverse estensioni delle API C:

- Message-Passing Interface (MPI) [Library]
- Posix Threads (Pthreads) [Library]
- OpenMP [Library + Compiler]
- CUDA [Library + Compiler]

Esistono librerie di livello superiore, ma si scambiano con facilità d'uso per le prestazioni (cioè, più performante/efficiente vuoi essere, più devi soffrire).

Perché abbiamo bisogno di quattro librerie diverse?

- Memoria condivisa
 - I core condividono la stessa memoria
 - Possono comunicare tra loro scrivendo e leggendo la stessa memoria
- Memoria distribuita
 - I core hanno la propria memoria
 - Devono comunicare tra loro inviando messaggi

Tipi di sistemi paralleli

- Multiple Instruction, Multiple Data (MIMD)
 - Ogni core ha le sue unità di controllo e lavora indipendentemente dagli altri core.
- Single Instruction, Multiple Data (SIMD) (e.g., GPUs)
 - I core condividono la stessa unità di controllo (devono eseguire la stessa istruzione oppure stare inattivi).

Come scriviamo programmi paralleli?

	Memoria condivisa	Memoria distribuita
SIMD	CUDA	
MIMD	Pthreads/ OpenMP/ CUDA	MPI

Concorrenza vs. Parallelismo vs. Distribuzione

- **Concorrenza:** Più attività possono essere in corso in qualsiasi momento.
 - **Parallelismo:** Compiti multipli cooperano strettamente per risolvere un problema.
 - **Distribuzione:** Un programma potrebbe aver bisogno di collaborare con altri programmi per risolvere un problema.
-
- Parallelo e distribuito sono concorrenti.
 - Programmi concorrenti possono essere seriali (e.g., sistemi operativi multitasking che girano su un singolo core).
 - Parallelo: strettamente accoppiato (i core condividono la memoria).
 - Distribuito: debolmente accoppiato (i core non condividono la memoria).

2 Memoria Distribuita: programmare con MPI

2.1 Sistemi a memoria distribuita

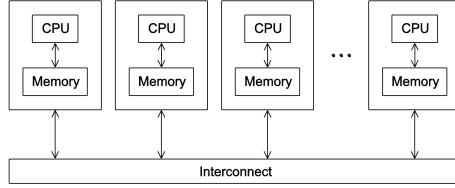


Figure 1: Esempio di sistema a memoria distribuita

2.2 Single Program Multiple Data (SPMD)

- Compiliamo un programma.
- Lo stesso programma è eseguito da più core.
- Usi if-else per specificare cosa deve fare ogni processo (simile a quello che succede quando fork un processo).
- Esempio. If i am process number 0, do X. If i am process number 1, do Y.
- I processi non condividono la memoria, quindi le comunicazioni avvengono attraverso il passaggio di messaggiI processi non condividono la memoria, quindi le comunicazioni avvengono attraverso il passaggio di messaggi.

2.3 MPI

Hello World in MPI

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello world from process %d\n", rank);
    MPI_Finalize();
    return 0;
}
```

MPI Components

- **MPI_Init:** Inizializza MPI.

```
– int MPI_Init(int *argc /* in/out */,
               char ***argv /* in/out */);
```

- **MPI_Comm_rank:** Restituisce il rango del processo chiamante.

```
– int MPI_Comm_rank(MPI_Comm comm /* in */,
                     int *rank /* out */);
```

- **MPI_Comm_size:** Restituisce il numero di processi in un comunicatore.

```
– int MPI_Comm_size(MPI_Comm comm /* in */,
                     int *size /* out */);
```

- **MPI_Finalize:** Termina MPI.

```
– int MPI_Finalize(void);
```

Compilazione.

```
mpicc -g -Wall -o hello hello.c
```

Esecuzione.

```
mpirun -n <number of processes> ./hello
```

Comunicatore.

Una collezione di processi può comunicare tra loro utilizzando un comunicatore. MPI_Init definisce un comunicatore che consiste in tutti i processi creati quando il programma è startato. Il comunicatore predefinito è chiamato MPI_COMM_WORLD.

Hello World! (v1)

```
mpirun -n 4 ./hello

Hello world from process 3
Hello world from process 1
Hello world from process 2
Hello world from process 0
```

Nota: L'ordine di stampa non è garantito.

MPI_Send e MPI_Recv

- **MPI_Send:** Invia un messaggio.

```
– int MPI_Send(void* buf /* in */ buffer,  
              int count /* in */ numero di elementi,  
              MPI_Datatype datatype /* in */ tipo di dato,  
              int dest /* in */ destinatario,  
              int tag /* in */ settiamo a NULL,  
              MPI_Comm comm /* in */);
```

- **MPI_Recv:** Riceve un messaggio.

```
– int MPI_Recv(void* buf /* out */ buffer,  
               int count /* in */ numero di elementi,  
               MPI_Datatype datatype /* in */ tipo di dato,  
               int source /* in */ mittente,  
               int tag /* in */ settiamo a NULL,  
               MPI_Comm comm /* in */,  
               MPI_Status *status /* out */);
```

Hello World! (v2)

Hello World! (v2)

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

const int MAX_STRING = 100;

int main(void){
    char greeting[MAX_STRING];
    int comm_sz;
    int my_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0){
        sprintf(greeting, "Greetings from process %d of %d!",
                my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
                 MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n",
               my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++){
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }
    MPI_Finalize();
    return 0;
}
```

```
mpirun -n 4 ./hello
```

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

Sending order

- **Blocking:** MPI_Send blocca fino a quando il messaggio non è stato inviato.
- **Non-blocking:** MPI_Isend restituisce immediatamente.

Datatype	Description
MPI_CHAR	Character
MPI_INT	Integer
MPI_FLOAT	Floating point
MPI_DOUBLE	Double precision floating point
MPI_BYTE	Byte
MPI_PACKED	Packed data
MPI_SHORT	Short integer
MPI_LONG	Long integer
MPI_UNSIGNED_CHAR	Unsigned character
MPI_UNSIGNED_SHORT	Unsigned short integer
MPI_UNSIGNED_LONG	Unsigned long integer
MPI_UNSIGNED	Unsigned integer
MPI_LONGDOUBLE	Long double precision floating point

Table 1: Common MPI Datatypes

- **Buffered:** MPI_Bsend invia il messaggio in un buffer.
- **Synchronous:** MPI_Ssend blocca fino a quando il messaggio non è stato ricevuto.
- **Ready:** MPI_Rsend invia un messaggio solo se il destinatario è pronto.
- **Collective:** MPI_Gather, MPI_Scatter, MPI_Allgather, MPI_Alltoall.

Quanti dati sto ricevendo?

```
int MPI_Get_count(
    MPI_Status *status      /* in */,
    MPI_Datatype datatype   /* in */,
    int *count              /* out */);
```

WARNINGS

- Se un processo tenta di ricevere un messaggio e non c'è un invio corrispondente, il processo si bloccherà.
- Se una chiamata a MPI_Send si blocca e non c'è ricezione corrispondente, il processo di invio può bloccarsi.
- Se una chiamata a MPI_Send è buffering e non c'è ricezione corrispondente, il messaggio andrà perso.
- Se il rango del processo di destinazione è lo stesso del processo di origine, un processo si bloccherà o, forse peggio, la ricezione può corrispondere a un altro invio.

2.4 Parallel Program Design

Foster's Methodology

- **Partitioning:** Dividere il calcolo da eseguire e i dati gestiti dal calcolo in piccole attività. L'attenzione qui dovrebbe essere sull'identificazione delle attività che possono essere eseguite in parallelo.
- **Communication:** Determinare quale comunicazione deve essere effettuata tra i compiti identificati nel passaggio precedente.
- **Agglomeration or aggregation:** Combinare le attività e le comunicazioni identificate nel primo passaggio in compiti più grandi. Ad esempio, se l'attività A deve essere eseguita prima che l'attività B possa essere eseguita, potrebbe avere senso aggregare in una singola attività composita.
- **Mapping:** Assegna le attività composite identificate nel passaggio precedente a processi/thread. Questo dovrebbe essere fatto in modo che la comunicazione sia ridotta al minimo e ogni processo/thread riceva all'incirca la stessa quantità di lavoro.

2.5 Parallel Design Patterns

Program Structure Patterns

Possiamo distinguere le attività in due categorie:

- **Globally Parallel, Locally Sequential (GPLS):** Significa che le applicazioni sono in grado di eseguire attività in parallelo, con ogni attività che esegue task sequenzialmente. I pattern che ricadono in questa categoria sono:
 - Single Program Multiple Data (SPMD)
 - Multiple Program Multiple Data (MPMD)
 - Master-Worker
 - Map-Reduce
- **Globally Sequential, Locally Parallel (GSLP):** Significa che le applicazioni eseguono un programma sequenziale, con le attività che eseguono task in parallelo. I pattern che ricadono in questa categoria sono:
 - Fork-Join
 - Loop Parallelism

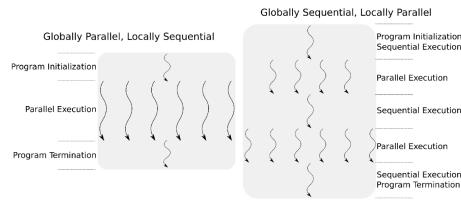


Figure 2: Esempio di Programmazione Parallela

Single Program Multiple Data (SPMD)

Tiene tutta la logica di controllo in un singolo programma. La struttura del programma contiene:

- **Program initialization:** e.g. inizializzazioni a runtime.
- **Obtain a unique identifier:** Gli identificatori sono numerati da 0, enumerando i thread o i processi utilizzati. Alcuni sistemi utilizzano identificatori vettoriali (ad es. CUDA).
- **Running the program:** Percorso di esecuzione diversificato in base all'ID.
- **Shutting-down the program:** Pulizia, salvataggio dei risultati, ecc.

Multiple Program Multiple Data (MPMD)

SPMD fallisce quando la memoria condivisa non è sufficiente. In MPMD, ogni programma ha la propria memoria.

L'esecuzione di un programma MPMD è simile a quella di un programma SPMD. Ma la distribuzione coinvolge programmi diversi. Sia SPMD che MPMD sono supportati da MPI.

Master-Worker

Due tipi di componenti: Master e Worker. Il Master(uno o più) è responsabile della suddivisione del lavoro e della raccolta dei risultati, ma anche dell'interazione con l'utente. I Master possono essere un bottleneck. I Worker sono responsabili dell'esecuzione del lavoro.

Esempio: Il Master invia un'immagine a ciascun Worker, che la elabora e restituisce il risultato al Master.

Map-Reduce

Variazione del Master-Worker. Il Master coordina tutta l'operazione. I Worker eseguono due tipi di operazioni: Map e Reduce.

- **Map:** Applica una funzione a ciascun elemento di un insieme di dati.
- **Reduce:** Combina i risultati di Map in un singolo risultato.

Parallel Design Patterns

Master-Worker: Stessa funzione applicata a differenti dati.

Map-Reduce: Stessa funzione applicata a differenti parti dello stesso dato.

Fork-Join

Un singolo thread esegue un'attività principale. Quando raggiunge un punto di fork, il thread si divide in due o più thread. Quando tutti i thread hanno completato il loro lavoro, si uniscono di nuovo. Utilizzato in OpenMP/Pthread.

Esempio:

```
mergesort(A, lo, hi):
    if lo < hi:                                // at least one element of input
        mid = (lo + hi) / 2
        fork mergesort(A, lo, mid)    //process(potentially) in
                                       //parallel with main task
        mergesort(A, mid, hi)      //main task handles
                                       //second recursion
        join
    merge(A, lo, mid, hi)
```

Loop Parallelism

Impiegato per la migrazione di software legacy/sequenziale a multicore. Si concentra sulla rottura dei cicli manipolando la variabile di controllo del ciclo. Un ciclo deve essere in una forma particolare per supportarlo. Flessibilità limitata, ma anche sforzo di sviluppo limitato.

Supportato da OpenMP.

2.6 Check for completion

Blocking (destroys handle)

```
int MPI_Wait(MPI_Request *request /* in/out */,
              MPI_Status *status /* out */);
```

Non-blocking (destroy handle if operation was successfull)

```
int MPI_Test(MPI_Request *request /* in/out */,
             int *flag /* out */,
             MPI_Status *status /* out */);
```

Ci sono diverse varianti disponibili:

- **MPI_Waitall:** Aspetta che tutti i messaggi siano completati.
- **MPI_Waitany:** Aspetta che uno qualsiasi dei messaggi sia completato.
- **MPI_Waitsome:** Aspetta che un certo numero di messaggi sia completato.
- **MPI_Testall:** Testa se tutti i messaggi sono completati.

- **MPI_Testany:** Testa se uno qualsiasi dei messaggi è completato.
- **MPI_Testsome:** Testa se un certo numero di messaggi è completato.

Esempio di Non-Blocking:

- *Problem: Ring:* Each rank sends something to left/right rank, and receives something from them

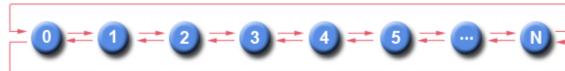


Figure 3: Esempio di Non-Blocking

Non-Blocking

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

int main(void){
    int numtasks, rank, next, prev, buf[2];
    MPI_Request reqs[4];
    MPI_Status stats[4];
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,
                  &rank); //determina i vicini sx e dx
    prev = (rank - 1) % numtasks;
    next = (rank + 1) % numtasks;
    // post non-blocking receives and sends for neighbors
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, 0,
              MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, 1,
              MPI_COMM_WORLD, &reqs[1]);
    MPI_Isend(&rank, 1, MPI_INT, prev, 1,
              MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, 0,
              MPI_COMM_WORLD, &reqs[3]);
    // do some work while sends/receives progress

    // wait for all non-blocking operations to complete
    MPI_Waitall(4, reqs, stats);
    // continue on...
    MPI_Finalize();
    return 0;
}
```

Input

Molte implelmentazioni MPI consentono solo al processo 0 in MPI_COMM_WORLD di accedere allo stdin.

Il processo 0 deve leggere i dati (scanf) e inviarli agli altri processi.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_input(rank, comm_sz, &n, &local_n, &local_a, &local_b);

h = (b - a) / n;
...

void Get_input(int my_rank, int comm_sz, int* n_p, int* local_n_p,
               double* a_p, double* b_p){
    int dest;
    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++){
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

2.7 Collective Communication

MPI_Reduce



Figure 4: Esempio di MPI_Reduce

MPI_Reduce

```
int MPI_Reduce(const void* sendbuf      /* in */,
               void* recvbuf       /* out */,
               int count           /* in */,
               MPI_Datatype datatype /* in */,
               MPI_Op op           /* in */,
               int root             /* in */,
               MPI_Comm comm        /* in */);
```

```
MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
```

```
double local_sum, total_sum;
...
MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
```

Regola del trapezio

```
int main(void){
    int my_rank, comm_sz, local_n;
    double a, b, h, local_a, local_b;
    double local_int, total_int;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    Get_input(my_rank, comm_sz, &a, &b, &n);

    h = (b - a) / n; // h is the same for all processes
    local_n = n / comm_sz; //so is the number of trapezoids

    /* Length of each process' interval of
       integration = local_n*h. */
    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);

    // Reduce the local integrals to the global integral
    MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);

    if (my_rank == 0){
        printf("With n = %d trapezoids, our estimate\n",
               n);
        printf("of the integral from %f to %f = %.15e\n",
               a, b, total_int);
    }
    MPI_Finalize();
    return 0;
}
```

MPI_Reduce Operators

Operator	Description
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bitwise XOR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

MPI_Bcast

MPI_Bcast

```
int MPI_Bcast(void* buffer           /* in/out */,
              int count            /* in */,
              MPI_Datatype datatype /* in */,
              int root              /* in */,
              MPI_Comm comm         /* in */);
```

Possiamo modificare il codice precedente che mostrava come mandare input dal rank 0 agli altri rank.

```
...
Get_input(my_rank, comm_sz, &a, &b, &n);
...
void Get_input(int my_rank, int comm_sz, double* a_p,
               double* b_p, int* n_p){
    int dest;
    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

MPI_Allreduce

Concettualmente un MPI_Reduce seguito da un MPI_Bcast.

MPI_Allreduce

```
int MPI_Allreduce(const void* sendbuf /* in */,
                  void* recvbuf      /* out */,
                  int count          /* in */,
                  MPI_Datatype datatype /* in */,
                  MPI_Op op          /* in */,
                  MPI_Comm comm      /* in */);
```

Gli argomenti sono gli stessi di MPI_Reduce, ad eccezione di dest_process (non c'è) perché tutti i processi avranno il risultato.

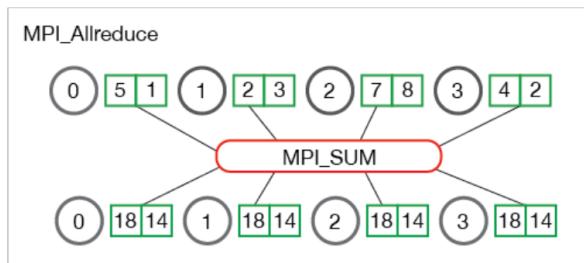


Figure 5: Esempio di MPI_Allreduce

Data una collettiva (ad esempio, MPI_Reduce), come selezionare l'algoritmo migliore?
Automaticamente attraverso l'euristica.

Manualmente: le implementazioni MPI come Open MPI non fanno ipotesi sull'hardware sottostante.

MPI_Scatter

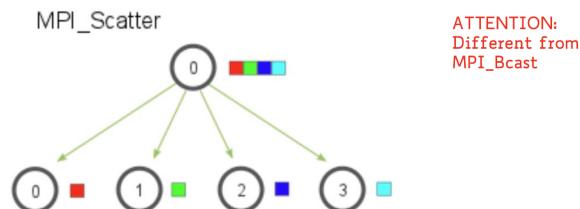


Figure 6: Esempio di MPI_Scatter

MPI_Scatter può essere utilizzata in una funzione che legge in input un intero vettore sul processo 0 m

MPI_Scatter

```
int MPI_Scatter(const void* sendbuf      /* in */,
                int sendcount        /* in */,
                MPI_Datatype sendtype /* in */,
                void* recvbuf        /* out */,
                int recvcount        /* in */,
                MPI_Datatype recvtype /* in */,
                int root             /* in */,
                MPI_Comm comm        /* in */);
```

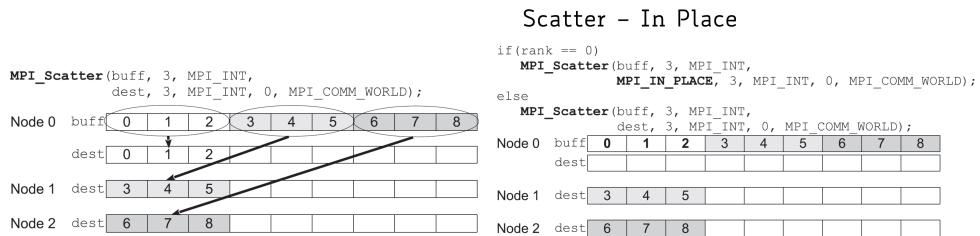


Figure 7: Esempi di utilizzo di MPI_Scatter

Reading and distributing a vector

```
void Read_vector(double local_a[], int local_n, int n,
                  int my_rank, int comm_sz){
    double* a = NULL;
    int i;
    if (my_rank == 0){
        a = malloc(n * sizeof(double));
        printf("Enter the vector\n");
        for (i = 0; i < n; i++){
            scanf("%lf", &a[i]);
        }
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a,
                    local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        free(a);
    }else{
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a,
                    local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
}
```

MPI_Gather

Raccoglie tutti i componenti del vettore sul processo 0, e quindi il processo 0 può elaborare tutti i com-

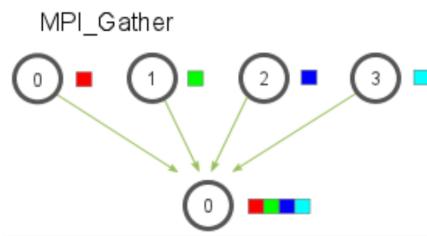


Figure 8: Esempio di MPI_Gather

MPI_Gather

```
int MPI_Gather(const void* sendbuf      /* in */,
               int sendcount        /* in */,
               MPI_Datatype sendtype /* in */,
               void* recvbuf        /* out */,
               int recvcount        /* in */,
               MPI_Datatype recvtype /* in */,
               int root              /* in */,
               MPI_Comm comm         /* in */);
```

sendcount è il numero di elementi che ogni processo invia, non il numero totale di elementi nel vettore finale!!!

Print a distributed vector

```
void Print_vector(double local_b[], int local_n, int n,
                  int my_rank, int comm_sz){
    double* b = NULL;
    int i;
    if (my_rank == 0){
        b = malloc(n * sizeof(double));
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b,
                   local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        printf("The vector is\n");
        for (i = 0; i < n; i++){
            printf("%f ", b[i]);
        }
        printf("\n");
        free(b);
    }else{
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b,
                   local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
}
```

MPI_Barrier

Un processo attende fino a quando tutti i processi nel comunicatore non raggiungono la barriera (da quel punto in poi i processi partono insieme, li "sincronizzo").

MPI_Barrier

```
int MPI_Barrier(MPI_Comm comm /* in */);
```

MPI_Allgather

Concettualmente è come fare un MPI_Gather seguito da un MPI_Bcast. In pratica potrebbe essere implementato più efficientemente (fa tutto da solo).

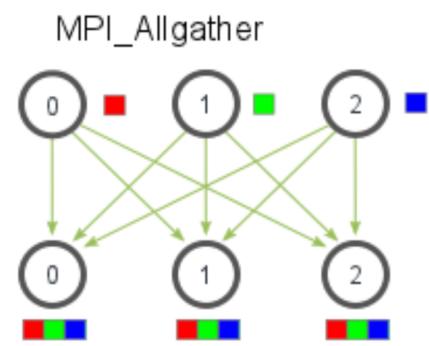


Figure 9: Esempio di MPI_Allgather

MPI_Allgather

```
int MPI_Allgather(const void* sendbuf    /* in */,
                  int sendcount      /* in */,
                  MPI_Datatype sendtype /* in */,
                  void* recvbuf     /* out */,
                  int recvcount     /* in */,
                  MPI_Datatype recvtype /* in */,
                  MPI_Comm comm     /* in */);
```

Reduce - Scatter



Figure 10: Esempio di Reduce - Scatter

MPI_Reduce_Scatter

```
int MPI_Reduce_scatter(const void* sendbuf    /* in */,
                      void* recvbuf      /* out */,
                      const int recvcounts[] /* in */,
                      MPI_Datatype datatype /* in */,
                      MPI_Op op          /* in */,
                      MPI_Comm comm      /* in */);
```

MPI_Alltoall

Ogni processo invia un messaggio a tutti gli altri processi.

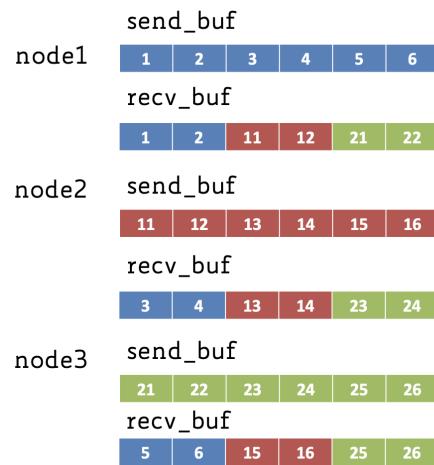


Figure 11: Esempio di MPI_Alltoall

MPI_Alltoall

```
int MPI_Alltoall(const void* sendbuf    /* in */,
                  int sendcount     /* in */,
                  MPI_Datatype sendtype /* in */,
                  void* recvbuf     /* out */,
                  int recvcount    /* in */,
                  MPI_Datatype recvtype /* in */,
                  MPI_Comm comm    /* in */);
```

2.8 Performance Evaluation

Elapsed parallel time

Elapsed parallel time

```
double start = MPI_Wtime();
...
double finish = MPI_Wtime();
printf("Elapsed time = %e seconds\n", finish - start);
```

Restituisce il numero di secondi trascorsi da qualche tempo nel passato.

Ogni rank potrebbe finire in un momento diverso. Proprio per questo riportiamo il tempo di esecuzione maggiore. **Come?**

```
double start = MPI_Wtime();
...
double finish = MPI_Wtime();
double elapsed = finish - start;
double max_elapsed;
MPI_Reduce(&elapsed, &max_elapsed, 1, MPI_DOUBLE,
           MPI_MAX, 0, MPI_COMM_WORLD);
if (my_rank == 0){
    printf("Max elapsed time = %e seconds\n", max_elapsed);
}
```

Ogni grado inizierà allo stesso tempo?

Non necessariamente, in caso contrario, il tempo che riportiamo potrebbe essere più lungo non perché l'applicazione stava funzionando male, ma piuttosto perché qualcuno ha iniziato più tardi di qualcun altro.

Come garantire che inizieranno allo stesso tempo?

MPI_Barrier

Domanda:

Ma aspetta ... MPI_Barrier è essa stessa una collettiva, e se è implementata con un albero? Ogni grado potrebbe uscire dalla barriera in un momento diverso

Risposta:

Garantire che inizino esattamente nello stesso momento è un po' più complicato. Ai fini di questo corso, MPI_Barrier fornisce un'approssimazione ragionevole.

È sufficiente una corsa/misurazione?

No, i dati sulle prestazioni non sono deterministici.

Soluzione:

eseguire l'applicazione più volte.

Cosa riportiamo?

Tempo minimo, tempo massimo, media?

Soluzione:

riporta l'intera distribuzione dei tempi.

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

The runtime increases with the problem size

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

The runtime decreases with the number of processes

Figure 12: Esempio di valutazione delle prestazioni

Performance Evaluation

Quali aspettative abbiamo?

Idealmente, quando viene eseguito con processi p, il programma dovrebbe essere p volte più veloce rispetto a quando viene eseguito con 1 processo.

Definiamo con $T_{serial}(n)$ il tempo della nostra applicazione sequenziale su un problema di dimensione n (ad esempio, n è la dimensione della matrice).

Definiamo con $T_{parallel}(n, p)$ il tempo della nostra applicazione parallela su un problema di dimensione n, quando si esegue con processi p.

Definiamo con $S(n, p)$ l'accelerazione (speedup) della nostra applicazione parallela.

$$S(n, p) = \frac{T_{serial}(n)}{T_{parallel}(n, p)}$$

Quindi, idealmente, vorremmo avere $S(n, p) = p$. In questo caso, diciamo che il nostro programma ha un'accelerazione lineare (linear speedup).

In generale, ci aspettiamo che l'accelerazione (speedup) migliori quando si aumenta la dimensione del problema n.

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Figure 13: Esempio di accelerazione lineare

ATTENZIONE!!!

$$T_{serial}(n)! = T_{parallel}(n, 1)$$

$T_{\text{serial}}(n)$ è il tempo della nostra applicazione sequenziale su un problema di dimensione n (ad esempio, n è la dimensione della matrice).

$T_{\text{parallel}}(n, 1)$ è il tempo della nostra applicazione parallela su un problema di dimensione n, quando viene eseguito con 1 processo.

Queste due implementazioni potrebbero essere diverse. In generale, $T_{\text{parallel}}(n, 1) \geq T_{\text{serial}}(n)$.

$$\begin{array}{ll} \text{Speedup} & \text{Scalability} \\ S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)} & S(n, p) = \frac{T_{\text{parallel}}(n, 1)}{T_{\text{parallel}}(n, p)} \end{array}$$

Efficiency

Efficiency

Efficiency è una misura di quanto bene un programma parallelo utilizza le risorse disponibili.

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \cdot T_{\text{parallel}}(n, p)}$$

Idealmente, vorremmo avere $E(n, p) = 1$. In pratica, è ≤ 1 , e peggiora con problemi di dimensioni più piccole.

Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Figure 14: Esempio di efficienza

Strong vs. Weak Scaling

Strong vs. Weak Scaling

Strong Scaling: Aumentare il numero di processi per risolvere un problema di dimensione fissa. Se possiamo mantenere un'alta efficienza, il nostro programma è fortemente scalabile.

Weak Scaling: Aumentare il numero di processi e la dimensione del problema in modo che la dimensione del problema per processo rimanga costante. Se possiamo mantenere un'elevata efficienza, il nostro programma è debole scalabile.

**Strong vs. Weak Scaling
(From Speedup Data)**

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Not strongly scalable

**Strong vs. Weak Scaling
(From Speedup Data)**

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Weakly scalable

Figure 15: Esempio di Strong e Weak Scaling su speedup data

**Strong vs. Weak Scaling
(From Efficiency Data)**

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Not strongly scalable

**Strong vs. Weak Scaling
(From Efficiency Data)**

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Weakly scalable

Figure 16: Esempio di Strong e Weak Scaling su efficiency data

Amdahl's Law

Amdahl's Law

Intuizione: Ogni programma ha alcune parti che non possono essere parallelizzate. Questa legge ci dice che lo speedup è limitato dalla frazione del programma che non può essere parallelizzata.

$$T_{parallel}(p) = (1 - \alpha) \cdot T_{serial} + \alpha \cdot \frac{T_{serial}}{p}$$

La frazione $0 \leq \alpha \leq 1$ è la parte del programma che può essere parallelizzata. Il rimanente $1 - \alpha$ è la parte che non può essere parallelizzata.

Esempio. se $\alpha = 0.9$, allora il 90% del programma può essere parallelizzato. $T_{parallel}(p) = \frac{T_{serial}}{p}$ è lo speedup ideale.

Possiamo estrapolare aspettative?

$$S(p) = \frac{T_{serial}}{(1 - \alpha) \cdot T_{serial} + \alpha \cdot \frac{T_{serial}}{p}}$$

$$\lim_{p \rightarrow \infty} S(p) = \frac{1}{1 - \alpha}$$

Esempio. se il 60% del programma può essere parallelizzato, $\alpha = 0.6$, allora lo speedup massimo è 2.5. (Massimo dello speedup, n. core massimi utilizzabili).

Gustafson's Law

Gustafson's Law

Se consideriamo lo weak scaling, la frazione parallela del programma aumenta con la dimensione del problema. (scaled speedup)

$$S(n, p) = (1 - \alpha) + \alpha \cdot p$$

Amdahl's Law vs. Gustafson's Law

Strong scaling weak scaling

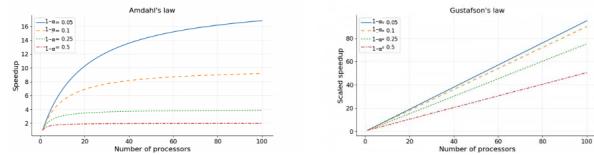


Figure 17: Amdahl's Law vs. Gustafson's Law

A parallel sorting algorithm

Parallel Sorting Algorithm

Serial bubble sort:

```
void Bubble_sort(int a[], int n){  
    int list_length, i, temp;  
  
    for (list_length = n; list_length >= 2; list_length--){  
        for (i = 0; i < list_length - 1; i++){  
            if (a[i] > a[i+1]){  
                temp = a[i];  
                a[i] = a[i+1];  
                a[i+1] = temp;  
            }  
        }  
    }  
}
```

Notiamo che non abbiamo molta opportunità di parallelizzare questo algoritmo.

Odd-even trasposition sort:

Una sequenza di fasi pari e dispari. In ogni fase, i processi scambiano i dati con i loro vicini.

Fase pari, compara e scambia:

$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \dots$

Fase dispari, compara e scambia:

$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \dots$

```
Start: 5, 9, 4, 3  
Even phase: compare-swap (5,9) and (4,3)  
            getting the list 5, 9, 3, 4  
Odd phase: compare-swap (9,3)  
            getting the list 5, 3, 9, 4  
Even phase: compare-swap (5,3) and (9,4)  
            getting the list 3, 5, 4, 9  
Odd phase: compare-swap (5,4)  
            getting the list 3, 4, 5, 9
```

Figure 18: Odd-even transposition sort esempio

Serial odd-even transposition sort

Serial Odd-Even Transposition Sort

```

void Odd_even_sort(int a[], int n){
    int phase, i, temp;
    for (phase = 0; phase < n; phase++){
        if (phase % 2 == 0){
            for (i = 1; i < n; i += 2){
                if (a[i-1] > a[i]){
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
            }
        } else{
            for (i = 1; i < n - 1; i += 2){
                if (a[i] > a[i+1]){
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
            }
        }
    }
}

```

Se il numero di elementi da ordinare è ugiale al numero di processi, ogni processo comunica con il vicino sinistro o destro in base alla fase (pari o dispari). Se il numero di elementi è maggiore del numero di processi? I processi si scambiano la loro porzione di array con i vicini e il processo con il rank minore si terrà nella sua porzione gli elementi minori.

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Figure 19: Odd-even transposition sort esempio

Compute partner

Compute Partner

```
int Compute_partner(int phase, int my_rank, int p){  
    int partner;  
    if (phase % 2 == 0){  
        if (my_rank % 2 != 0){  
            partner = my_rank - 1;  
        }else{  
            partner = my_rank + 1;  
        }  
    }else{  
        if (my_rank % 2 != 0){  
            partner = my_rank + 1;  
        }else{  
            partner = my_rank - 1;  
        }  
    }  
    if (partner == -1 || partner == p){  
        partner = MPI_PROC_NULL;  
    }  
    return partner;  
}
```

Parallel odd-even transposition sort

Parallel Odd-Even Transposition Sort

```
void Merge_low(int local_a[], int temp_b[], int temp_c[],  
              int local_n){  
    int ai, bi, ci;  
    ai = bi = ci = 0;  
    while (ci < local_n){  
        if (temp_b[bi] <= local_a[ai]){  
            local_a[ci] = temp_b[bi];  
            ci++; bi++;  
        }else{  
            local_a[ci] = local_a[ai];  
            ci++; ai++;  
        }  
    }  
    for (ai = 0; ai < local_n; ai++){  
        local_a[ai] = temp_c[ai];  
    }  
}
```

MPI_Sendrecv

MPI_Sendrecv

```
int MPI_Sendrecv(const void* sendbuf /* in */,
                 int sendcount /* in */,
                 MPI_Datatype sendtype /* in */,
                 int dest /* in */,
                 int sendtag /* in */,
                 void* recvbuf /* out */,
                 int recvcount /* in */,
                 MPI_Datatype recvtype /* in */,
                 int source /* in */,
                 int recvtag /* in */,
                 MPI_Comm comm /* in */,
                 MPI_Status *status /* out */);
```

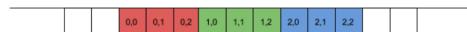
E' come utilizzare MPI_Send e MPI_Recv in sequenza, ma in un'unica chiamata. La destinazione e la sorgente possono essere lo stesso processo. Particolarmente utile perché MPI pianifica le comunicazioni in modo che il programma non si blocchi o crashi.

2.9 Prodotti di matrici e vettori in MPI

Gather/Broadcast su matrici

```
int matrix[3][3];
```

row,col	0,0	0,1	0,2
	1,0	1,1	1,2
	2,0	2,1	2,2



```
MPI_Reduce(sendbuf, recvbuf, num_rows*num_cols,
            MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

Figure 20: Esempio di Gather/Broadcast su matrici

Gather/Broadcast su matrici dinamiche

Gather/Broadcast su matrici dinamiche

Abbiamo un array di puntatori, ogni elemento è un puntatore ad una riga della matrice.

```
int** a;
a = (int**) malloc(sizeof(int*)*num_rows),
for (i = 0; i < num_rows; i++){
    a[i] = (int*) malloc(sizeof(int)*num_cols);
}
```

Per utilizzare MPI_Reduce:

```
for(i = 0; i < num_rows; i++){
    MPI_Reduce(a[i], b[i], num_cols, MPI_INT, MPI_SUM, 0,
               MPI_COMM_WORLD);
}
```

In questo caso abbiamo una matrice allocata come array.

```
Gather/Broadcast on dynamically allocated matrices
int* a;
a = (int*) malloc(sizeof(int)*num_rows*num_cols);
...
...
// a[i][j]
a[i * num_cols + j] = ....
```

CORRECT

Esempio 1: Somma tra vettori

Esempio 1: Somma tra vettori parallela

```
void Parallel_vector_sum(const int local_A[],
                         const int local_B[], int local_C[], int local_n){

    int local_i;

    for (local_i = 0; local_i < local_n; local_i++){
        local_C[local_i] = local_A[local_i] + local_B[local_i];
    }
}
```

Esempio 1: Stampa un vettore distribuito

```
void Print_vector(const int local_A[], int local_n,
                  int n, char title[], int my_rank,
                  MPI_Comm comm){
    int* A = NULL;
    int i;
    if (my_rank == 0){
        A = malloc(n * sizeof(int));
        MPI_Gather(local_A, local_n, MPI_INT, A,
                   local_n, MPI_INT, 0, comm);
        printf("%s\n", title);
        for (i = 0; i < n; i++){
            printf("%d ", A[i]);
        }
        printf("\n");
        free(A);
    }else{
        MPI_Gather(local_A, local_n, MPI_INT, A,
                   local_n, MPI_INT, 0, comm);
    }
}
```

Esempio 2: Prodotto matrice-vettore

Matrix Scattering

```
void Read_matrix(char prompt[], int local_A[], int m,
                  int n, int local_m, int my_rank,
                  MPI_Comm comm){
    int i, j;
    int* A = NULL;
    local_ok = 1;

    if(my_rank == 0){
        A = malloc(m * n * sizeof(int));
        printf("%s\n", prompt);
        for (i = 0; i < m; i++){
            for (j = 0; j < n; j++){
                scanf("%d", &A[i*n+j]);
            }
        }
        MPI_Scatter(A, local_m*n, MPI_INT, local_A,
                   local_m*n, MPI_INT, 0, comm);
        free(A);
    }else{
        MPI_Scatter(A, local_m*n, MPI_INT, local_A,
                   local_m*n, MPI_INT, 0, comm);
    }
}
```

Come parallelizzarlo?

Ora, supponiamo che questo sia fatto in un ciclo e che l'output y sia usato come vector di input per la prossima iterazione.

e.g.:

- Iteration 0: $y_0 = Ax_0$
- Iteration 1: $y_1 = Ax_1$
- Iteration 2: $y_2 = Ax_2$
- ...

Prima iterazione:

1. Broadcast il vettore x a tutti i processi.
2. Scatter le righe della matrice dal rank 0 agli altri processi.
3. Ogni processo calcola la somma delle righe della matrice moltiplicate per il vettore x .
4. Gather i risultati parziali al rank 0.
5. Broadcast il risultato dal rank 0 agli altri processi.

I punti 4 e 5 si possono unire con una Allgather.

Iterazioni successive:

1. Ogni processo computa un sottoinsieme degli elementi risultante, utilizzando il vettore y del passo precedente.
2. Gather il vettore finale al rank 0.
3. Broadcast y dal rank 0 agli altri processi.

Anche in questo caso i punti 4 e 5 si possono unire con una Allgather.

Prodotto matrice-vettore

```
void Mat_vect_mult(double local_A[], double local_x[],
                    double local_y[], int local_m, int n,
                    int local_n, MPI_Comm comm){
    double* x;
    int local_i, j;
    int local_ok = 1;
    x = malloc(n * sizeof(double));
    MPI_Allgather(local_x, local_n, MPI_DOUBLE, x,
                  local_n, MPI_DOUBLE, comm);
    for (local_i = 0; local_i < local_m; local_i++){
        local_y[local_i] = 0.0;
        for (j = 0; j < n; j++){
            local_y[local_i] += local_A[local_i*n+j] * x[j];
        }
    }
    free(x);
}
```

2.10 MPI Derived Data Types

Utilizzato per rappresentare qualsiasi raccolta di elementi di dati in memoria memorizzando sia i tipi di elementi che le loro posizioni relative in memoria. L'idea è che se una funzione che invia dati conosce queste informazioni su una raccolta di elementi di dati, può raccogliere gli elementi dalla memoria prima che vengano inviati. Allo stesso modo, una funzione che riceve i dati può distribuire gli elementi nelle loro destinazioni corrette in memoria quando vengono ricevuti.

Comunicare strutture

Funziona? Si, ma potrebbe non funzionare(esempio. Sistema a 64 bit vs 32 bit).

```
const int N = ...;

typerdef struct Point{
    int x;
    int y;
    int color;
};

Point image[N];
...
MPI_Send(image, N*sizeof(Point), MPI_BYTE, dest, tag,
         MPI_COMM_WORLD);
```

MPI_Type_create_struct

MPI_Type_create_struct

```
int MPI_Type_create_struct(int count, int blocklens[],  
                           MPI_Aint displacements[],  
                           MPI_Datatype types[],  
                           MPI_Datatype *newtype);
```

e.g., per la struttura:

```
struct t{  
    double a;  
    double b;  
    int c;  
}
```

count = 3; blocklens = 1, 1, 1; displacements = 0, 8, 16; types = MPI_DOUBLE,
MPI_DOUBLE, MPI_INT;

Nel caso di una struttura diversa:

```
struct t{  
    double a[10];  
    int b;  
    double c;  
}
```

Come calcolo il displacement? **MPI_Get_address**

MPI_Get_address

```
int MPI_Get_address(const void* location, MPI_Aint *address);
```

Nel caso di una struttura:

```
struct t{  
    double a[10];  
    int b;  
    double c;  
}  
  
MPI_Aint a_addr, b_addr, c_addr;  
  
MPI_Get_address(&t.a, &a_addr);  
array_of_displacements[0] = 0;  
MPI_Get_address(&t.b, &b_addr);  
array_of_displacements[1] = b_addr - a_addr;  
MPI_Get_address(&t.c, &c_addr);  
array_of_displacements[2] = c_addr - a_addr;
```

MPI_Type_commit

MPI_Type_commit

```
int MPI_Type_commit(MPI_Datatype *datatype);
```

Consente all'implementazione MPI di ottimizzare la sua rappresentazione interna del tipo di dati per l'uso nelle funzioni di comunicazione.

MPI_Type_free

MPI_Type_free

```
int MPI_Type_free(MPI_Datatype *datatype);
```

Quando abbiamo finito con il nostro nuovo tipo, questo libera qualsiasi spazio di archiviazione aggiuntivo utilizzato.

Unendo tutto insieme:

```
void Build_mpi_type(
    double* a_p           /* in */,
    double* b_p           /* in */,
    int* n_p              /* in */,
    MPI_Datatype* input_mpi_t_p /* out */) {
    int array_of_blocklengths[3] = {1, 1, 1};
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint a_addr, b_addr, n_addr;
    MPI_Aint array_of_displacements[3] = {0};
    MPI_Get_address(a_p, &a_addr);
    MPI_Get_address(b_p, &b_addr);
    MPI_Get_address(n_p, &n_addr);
    array_of_displacements[1] = b_addr - a_addr;
    array_of_displacements[2] = n_addr - a_addr;
    MPI_Type_create_struct(3, array_of_blocklengths,
                          array_of_displacements, array_of_types,
                          input_mpi_t_p);
    MPI_Type_commit(input_mpi_t_p);
} /* Build_mpi_type */

void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
              int* n_p) {
    MPI_Datatype input_mpi_t;
    Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

    MPI_Type_free(&input_mpi_t);
} /* Get_input */
```

It is called MPI_Type_create_struct, but the elements must not be necessarily come from a struct (you can use any sequence of variables)

Figure 21: Esempio di MPI_Type_create_struct

3 Pthreads

WARNING!

DA QUI IN POI SOLO CON LINUX (FORSE ANCHE CON MAC, CI SONO ALTERNATIVE VALIDE).

Distributed Memory Systems

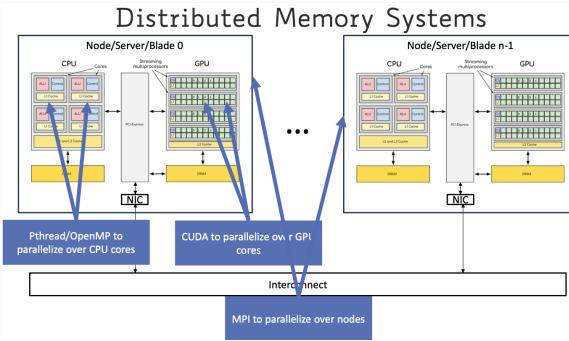


Figure 22: Esempio di sistema a memoria distribuita

Memory layout: process Memory layout: thread

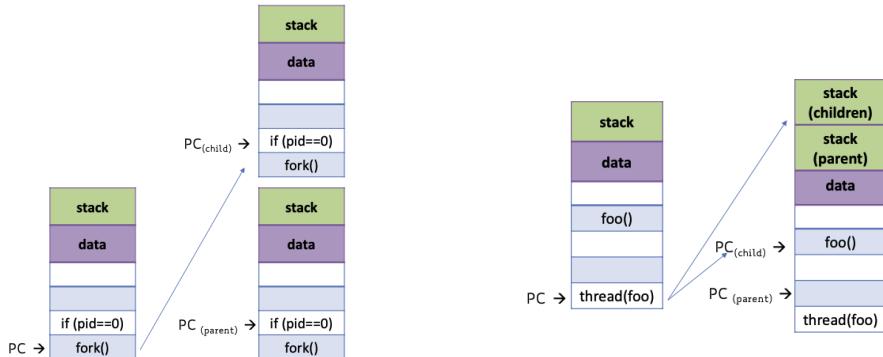


Figure 23: Differenza processo/thread

Pthread è uno standard POSIX per la programmazione concorrente. Specifica un'interfaccia di programmazione delle applicazioni (API) per la programmazione multi-thread.

Creazione di un thread

Creazione di un thread

```
#include <pthread.h>
int pthread_create(pthread_t *thread,           /* out */
                  const pthread_attr_t *attr,      /* in */
                  void *(*start_routine) (void *), /* in */
                  void *arg);                   /* in */;
```

il parametro attr sarà sempre settato a NULL nel nostro caso.

È un handle (uno per thread). Ad esempio, un "oggetto" che rappresenta un thread deve essere assegnato prima della chiamata (opaco). I dati effettivi che memorizzano sono specifici del sistema. I loro membri dei dati non sono direttamente accessibili al codice utente. Tuttavia, lo standard Pthreads garantisce che un oggetto pthread_t memorizza abbastanza informazioni per identificare in modo univoco il thread a cui è associato.

La funzione che il thread sta per eseguire è un puntatore a una funzione (l'indirizzo di un pezzo di memoria contenente codice piuttosto che dati) . In questo caso, abbiamo bisogno di una funzione che restituisce un void* e prende come argomento un void*.

Puntatore a funzione in C

Puntatore a funzione in C

```
#include <stdio.h>
#include <pthread.h>

void* my_function(void* arg){
    int* px = (int*) arg;
    int x = *px;
    printf("Hello from thread %d\n", x);
}

void main(){
    pthread_t thread;
    int x = 2;
    pthread_create(&thread, NULL, my_function, &x);
    ...
}
```

Variabili globali

Può introdurre bug sottili e confusi! Limitare l'uso delle variabili globali alle situazioni in cui sono davvero necessarie.

Aspettare la fine di un thread

Aspettare la fine di un thread

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

se il valore di retval non è NULL, pthread_join() memorizzerà il valore restituito dalla funzione del thread terminato in *retval.

Correct way to wait for thread completion

```
for(int i = 0; i < num_threads; i++){
    pthread_create(...);
}
for(int i = 0; i < num_threads; i++){
    pthread_join(...);
}
```

Correct

vs.

```
for(int i = 0; i < num_threads; i++){
    pthread_create(...);
    pthread_join(...);
}
```

Wrong
(everything would be
executed sequentially)

Figure 24: Modo corretto di fare create e join

Thread identification

pthread_self() restituisce l'identificatore del thread chiamante.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

pthread_equal() confronta due identificatori di thread.

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

Hello World con Pthreads

Hello World con Pthreads

```
#include <stdio.h>
#include <pthread.h>

void* my_function(void* arg){
    int* px = (int*) arg;
    int x = *px;
    printf("Hello from thread %d\n", x);
    return NULL;
}

void main(){
    pthread_t thread[4];
    int i;
    for (i = 0; i < 4; i++){
        pthread_create(&thread[i], NULL, my_function, &i);
    }
    for (i = 0; i < 4; i++){
        pthread_join(thread[i], NULL);
    }
}
```

per compilare:

```
gcc -o hello hello.c -lpthread
```

Nel caso di struct:

```
struct thread_args {
    long my_rank;
    char *task_name;
};

void *Hello(void *args) {
    struct thread_args* t_args
        = (struct thread_args *) args;
    printf("Thread %ld is working on task '%s'\n",
        t_args->my_rank, t_args->task_name);
    return NULL;
}

struct thread_args *t_args
    = malloc(sizeof(struct thread_args));
t_args->my_rank = thread;
t_args->task_name = "Hello task";
pthread_create(&thread_handles[thread],
    NULL,
    Hello,
    (void *) t_args);
```



Figure 25: Esempio di utilizzo di struct con Pthreads

3.1 Matrix - Vector Multiplication in Pthreads

Intuizione

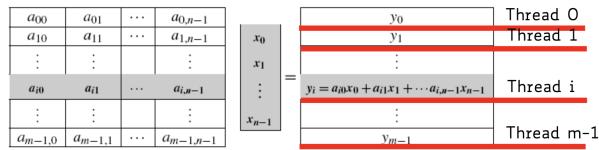


Figure 26: Esempio di moltiplicazione matrice-vettore con Pthreads

In linea di principio, dovresti cercare di evitare di avere più thread che core. Ci sono situazioni in cui potrebbe avere senso (ne discuteremo più avanti). Comunque, in generale dividerai le righe m su t thread, con $t < m$. Ognuno elabora le righe $\frac{m}{t}$

Il thread q processa le righe che partono da $q \cdot \frac{m}{t}$ fino a $(q + 1) \cdot \frac{m}{t} - 1$.

Nota!

Non abbiamo bisogno di fare scatter/broadcast, poiché ogni thread ha accesso a tutti i dati(stessa memoria).

Matrix - Vector Multiplication in Pthreads

```
void *Pth_mat_vect(void* rank){
    long my_rank = (long) rank;
    int i;
    int local_m = m/thread_count;
    int my_first_row = my_rank * local_m;
    int my_last_row = (my_rank + 1) * local_m - 1;
    for (i = my_first_row; i <= my_last_row; i++){
        y[i] = 0.0;
        for (j = 0; j < n; j++){
            y[i] += A[i][j] * x[j];
        }
    }
    return NULL;
}
```

Quanti thread dovrei usare?

In principio dovremmo evitare di avere più thread che core.

Come possiamo capire quanti core abbiamo?

```
$ lscpu | grep -E '^Thread|^Core|^Socket|^CPU\('
```

3.2 Sezioni critiche

Stima di Pi con Pthreads

Nel fare il calcolo dentro la funzione Thread_sum abbiamo una sezione critica. Per dare accesso esclusivo alla variabile sum, possiamo utilizzare un mutex.

Stima di Pi con Pthreads

```
int thread_count;
int n;
double sum;
pthread_mutex_t mutex;

void* Thread_sum(void* rank){
    int my_rank = (int) rank;
    double factor, my_sum = 0.0;
    int i;
    int my_n = n/thread_count;
    int my_first_i = my_n * my_rank;
    int my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0){
        factor = 1.0;
    }else{
        factor = -1.0;
    }
    for (i = my_first_i; i < my_last_i; i++, factor = -factor){
        my_sum += factor/(2*i+1);
    }
    pthread_mutex_lock(&mutex); //Sezione critica (lock)
    sum += my_sum; //Aggiorna la variabile condivisa uno per volta
    pthread_mutex_unlock(&mutex); //Fine sezione critica (unlock)
    return NULL;
}
int main(int argc, char* argv[]){
    int thread;
    pthread_t* thread_handles;
    thread_count = atoi(argv[1]);
    n = atoi(argv[2]);
    thread_handles = malloc(thread_count * sizeof(pthread_t));
    pthread_mutex_init(&mutex, NULL); //Inizializza il mutex
    sum = 0.0;
    for (thread = 0; thread < thread_count; thread++){
        pthread_create(&thread_handles[thread], NULL,
                      Thread_sum, (void*) thread);
    }
    for (thread = 0; thread < thread_count; thread++){
        pthread_join(thread_handles[thread], NULL);
    }
    sum = 4.0 * sum;
    printf("With n = %d terms,\n", n);
    printf("Our estimate of pi = %.15f\n", sum);
    pthread_mutex_destroy(&mutex); //Distrugge il mutex
    free(thread_handles);
    return 0;
}
```

Con i mutex dobbiamo stare attenti a possibili Deadlocks o Starvation.

Ci sono mutex non bloccanti (prova il lock, se riesce ok, altrimenti non si blocca ad aspettare).

```
pthread_mutex_trylock(&mutex);
```

Per i deadlock c'è il rischio che il thread A abbia lockato il mutex 1 e stia aspettando il mutex 2, mentre il thread B ha lockato il mutex 2 e sta aspettando il mutex 1.

3.3 Producer-consumer Synchronization and Semaphores

Busy-waiting impone l'ordine di accesso dei thread a una sezione critica. Usando i mutex, l'ordine è lasciato al caso e al sistema. Ci sono applicazioni in cui dobbiamo controllare l'ordine di accesso dei thread alla sezione critica.

Per esempio, nello scambio di messaggi ad anello (ricevi da sinistra e mandi a destra). Alcuni thread potrebbero leggere dati che non sono ancora stati scritti.
Introduciamo i semafori.

Producer-consumer Synchronization and Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

- `sem_wait (sem_t* sem)` blocca se il semaforo è 0. Se il semaforo è > 0 , decrementerà il semaforo e procederà.
- `sem_post (sem_t* sem)` se c'è un thread in attesa in `sem_wait()`, quel thread può procedere con l'esecuzione. Altrimenti, il semaforo viene incrementato.
- `sem_getvalue(sem_t* sem, int* sval)` pone il valore corrente del semaforo indicato a `sem` nell'intero indicato da `sval`.

- Mutex sono binari. Semafori sono **unsigned int**.
- Mutex iniziano unlockati. Semafori iniziano con un valore specificato.
- Mutex sono solitamente locked/unlocked dallo stesso thread. Semafori sono solitamente incrementati/decrementati da differenti thread.

3.4 Barriers and Condition Variables

Barriere

Barriere

Sincronizzare i thread per essere sicuri che sono tutti allo stesso punto del programma. Nessun thread può superare la barriera finché tutti i thread non sono arrivati.

Esempi di utilizzo

Time the slowest thread

```
/* Shared */
double elapsed_time;
...
/* Private */
double my_start, my_finish, my_elapsed;
...
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
...
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed_time = Maximum of my_elapsed values;
```

Implementare una barriera usando il busy-waiting e i mutex è semplice.

Usiamo un contatore condiviso protetto da un mutex.

Quando il contatore indica che tutti i thread sono arrivati, resettiamo il contatore e facciamo ripartire i thread.

Barriere con mutex

```
int counter; //Contatore condiviso settato a 0
int thread_count; //Numero di thread
pthread_mutex_t barrier_mutex; //Mutex per proteggere il contatore
...

void* Thread_work(void* rank){
    int my_rank = (int) rank;
    ...
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    ...
}
```

Se volessimo utilizzare la barriera di nuovo, abbiamo bisogno di resettare il contatore. Ma ciò potrebbe accadere prima dell'arrivo di tutti i thread.

Ora implementiamo la barriera utilizzando i semafori.

Barriere con semafori

```
sem_t barrier_sem; //Semaforo inizializzato a 0
sem_t count_sem; //Semaforo inizializzato a 1
int counter; //Contatore condiviso settato a 0
...
void* Thread_work(void* rank){
    int my_rank = (int) rank;
    // Barrier
    sem_wait(&count_sem);
    if (counter == thread_count - 1){
        counter = 0;
        sem_post(&count_sem);
        for (int j = 0; j < thread_count - 1; j++){
            sem_post(&barrier_sem);
        }
    }else{
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    ...
}
```

Qui abbiamo lo stesso problema di prima, se volessimo utilizzare la barriera di nuovo, abbiamo bisogno di resettare il contatore.

Per risolvere questo problema, introduciamo le **condition variables**.

Condition Variables

Condition Variables

Una condition variable è un oggetto che consente a un thread di attendere che un'altra attività notifichi che un evento è avvenuto. Quando l'evento o la condizione è soddisfatta, il thread può essere svegliato. La condition variable è sempre associata con un mutex.

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- wait rilascia il mutex e mette il thread in attesa.
- signal sveglia un thread in attesa.
- broadcast sveglia tutti i thread in attesa.
- destroy distrugge la condition variable.
- init inizializza la condition variable.

Ora implementiamo la barriera utilizzando le condition variables.

Barriere con condition variables

```
// Shared
int counter = 0;
pthread_mutex_t barrier_mutex;
pthread_cond_t cond_var;
void* Thread_work(void* rank){
    int my_rank = (int) rank;
    ... // Barriera
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    if (counter == thread_count){
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    }else{
        while (pthread_cond_wait(&cond_var, &barrier_mutex) != 0);
        //la forza a fare il Wait, potrebbe essere svegliata
        //da un segnale (spurious wake-up)
    }
    pthread_mutex_unlock(&barrier_mutex);
}
```

3.5 Read - Write Locks

Controllare l'accesso a una grande struttura di dati condivisa

Linked List

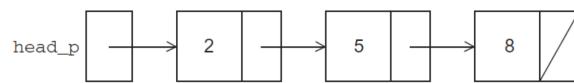


Figure 27: Esempio di Linked List

Linked List

```
struct list_node_s{
    int data;
    struct list_node_s* next;
};
```

Linked List Membership

```
int Member(int value, struct list_node_s* head_p){  
    struct list_node_s* curr_p = head_p;  
    while (curr_p != NULL && curr_p->data < value){  
        curr_p = curr_p->next;  
    }  
    if (curr_p == NULL || curr_p->data > value){  
        return 0;  
    }else{  
        return 1;  
    }  
}
```

Inserimento di un nodo

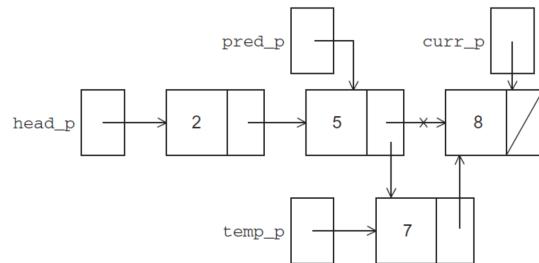


Figure 28: Esempio di inserimento di un nodo in una Linked List

Inserimento di un nodo

```
int Insert(int value, struct list_node_s** head_p){  
    struct list_node_s* curr_p = *head_p;  
    struct list_node_s* pred_p = NULL;  
    struct list_node_s* temp_p;  
    while (curr_p != NULL && curr_p->data < value){  
        pred_p = curr_p;  
        curr_p = curr_p->next;  
    }  
    if (curr_p == NULL || curr_p->data > value){  
        temp_p = malloc(sizeof(struct list_node_s));  
        temp_p->data = value;  
        temp_p->next = curr_p;  
        if (pred_p == NULL){ //value in testa  
            *head_p = temp_p;  
        }else{  
            pred_p->next = temp_p;  
        }  
        return 1;  
    }else{ //value in list  
        return 0;  
    }  
}
```

Rimozione di un nodo

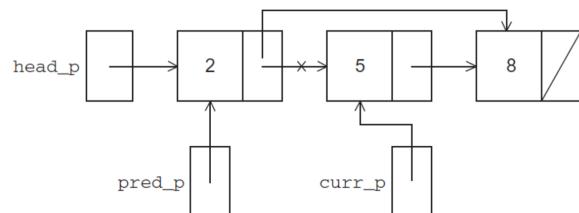


Figure 29: Esempio di rimozione di un nodo in una Linked List

Rimozione di un nodo

```
int Delete(int value, struct list_node_s** head_p){  
    struct list_node_s* curr_p = *head_p;  
    struct list_node_s* pred_p = NULL;  
    while (curr_p != NULL && curr_p->data < value){  
        pred_p = curr_p;  
        curr_p = curr_p->next;  
    }  
    if (curr_p != NULL && curr_p->data == value){  
        if (pred_p == NULL){  
            *head_p = curr_p->next;  
            free(curr_p);  
        }else{  
            pred_p->next = curr_p->next;  
            free(curr_p);  
        }  
        return 1;  
    }else{ //value non in list  
        return 0;  
    }  
}
```

A Multi - Threaded Linked List

Proviamo a usare queste funzioni in un programma Pthreads. Per condividere l'accesso alla lista, possiamo definire head_p come variabile globale. Questo semplificherà le intestazioni delle funzioni per Member, Insert ed Delete, poiché non avremo bisogno di passare né head_p né un puntatore a head_p: dovremo solo passare il valore di interesse. Se più thread chiamano Member allo stesso tempo, va bene. Cosa succede se un thread chiama Member mentre un altro thread sta eliminando un elemento?

Abbiamo due soluzioni:

1. lockare la lista tutte le volte che un thread ci accede. Utilizzando un mutex sulla lista intera.
 - Stiamo serializzando l'accesso. Se la maggior parte delle operazioni è di lettura, questo è un grosso spreco. Altrimenti è la soluzione migliore.
2. lockare i singoli nodi. Utilizzando un mutex per ogni nodo (lo inseriamo nella struct).
 - Se la lista è lunga, questo è un grosso spreco. E' molto più lenta della prima soluzione.

Un read - write lock è un po' come un mutex tranne per il fatto che fornisce due funzioni di blocco. La prima funzione di blocco blocca il blocco di lettura-scrittura per la lettura, mentre la seconda lo blocca per la scrittura.

Quindi più thread possono ottenere contemporaneamente il lock chiamando la funzione di read-lock, mentre solo un thread può ottenere il lock chiamando la funzione di write-lock.

Quindi, se qualche thread possiede il write-lock, qualsiasi thread che vuole ottenere il write-lock sarà bloccato nella chiamata alla funzione write-lock.

Se qualsiasi thread possiede il write-lock, tutti i thread che vogliono ottenere read-lock o write-lock saranno bloccati nelle rispettive funzioni di lock.

rwlock functions

pthread_rwlock functions

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                      const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Il secondo parametro di init possiamo lasciarlo a NULL. Utilizzandolo possiamo specificare chi ha la priorità tra lettori o scrittori.

Esempio di utilizzo

Esempio di utilizzo di pthread_rwlock

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
...
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
...
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

Linked List performance

Implementation	1	2	4	8	Number of Threads
Read-Write Locks	0.213	0.123	0.098	0.115	
One Mutex for Entire List	0.211	0.450	0.385	0.457	
One Mutex per Node	1.680	5.700	3.450	2.700	

100,000 ops/thread
99.9% Member
0.05% Insert
0.05% Delete
8 (physical) cores

Implementation	1	2	4	8	Number of Threads
Read-Write Locks	0.213	0.123	0.098	0.115	
One Mutex for Entire List	0.211	0.450	0.385	0.457	
One Mutex per Node	1.680	5.700	3.450	2.700	

100,000 ops/thread
99.9% Member
0.05% Insert
0.05% Delete
8 (physical) cores

Implementation	1	2	4	8	Number of Threads
Read-Write Locks	0.213	0.123	0.098	0.115	
One Mutex for Entire List	0.211	0.450	0.385	0.457	
One Mutex per Node	1.680	5.700	3.450	2.700	

100,000 ops/thread
99.9% Member
0.05% Insert
0.05% Delete
8 (physical) cores

Implementation	1	2	4	8	Number of Threads
Read-Write Locks	2.48	4.97	4.69	4.71	
One Mutex for Entire List	2.50	5.13	5.04	5.11	
One Mutex per Node	12.00	29.60	17.00	12.00	

100,000 ops/thread
80% Member
10% Insert
10% Delete
8 (physical) cores

Implementation	1	2	4	8	Number of Threads
Read-Write Locks	2.48	4.97	4.69	4.71	
One Mutex for Entire List	2.50	5.13	5.04	5.11	
One Mutex per Node	12.00	29.60	17.00	12.00	

100,000 ops/thread
80% Member
10% Insert
10% Delete
8 (physical) cores

Figure 30: Linked List Performance

3.6 Thread safety

Un blocco di codice è thread-safe se può essere eseguito contemporaneamente da più thread senza causare problemi. Supponiamo di voler usare più thread per "tokenizzare" un file che consiste in testo inglese ordinario. I token sono solo sequenze contigue di caratteri separate dal resto del testo da uno spazio bianco: uno spazio, una scheda o una nuova riga.

Dividi il file di input in righe di testo e assegna le righe ai thread in modo round-robin. La prima riga va al thread 0, la seconda va al thread 1, . . . , il t^{th} va al thread t , il $t+1^{st}$ va al thread 0, ecc.

Possiamo serializzare l'accesso alle linee di input usando i semafori.

Perché i semafori e non i mutex?

Dopo che un thread ha letto una singola riga di input, può tokenizzare la riga utilizzando la funzione strtok.

La funzione strtok

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

La prima volta che viene chiamato l'argomento stringa dovrebbe essere il testo da tokenizzare. La nostra linea di input. Per le chiamate successive, il primo argomento dovrebbe essere NULL.

L'idea è che nella prima chiamata, strtok memorizza nella cache un puntatore alla stringa e per le chiamate successive restituisca token successivi presi dal puntatore memorizzato nella cache.

Multi-threaded Tokenization

Multi-threaded Tokenization

```
sems[0] inizializzato a 1
sems[i] (con i > 0) inizializzato a 0

void* Tokenize(void* rank){
    int my_rank = (int) rank;
    int count;
    int next = (my_rank + 1) % thread_count;
    char* fg_rv; //fgets return value
    char my_line[MAX];
    char* my_string;
    sem_wait(&sems[my_rank]);
    fg_rv = fgets(my_line, MAX, stdin);
    sem_post(&sems[next]);
    while(fg_rv != NULL){
        printf("Thread %d > my line = %s", my_rank, my_line);
        count = 0;
        my_string = strtok(my_line, " \t\n");
        while (my_string != NULL){
            count++;
            printf("Thread %d > string %d = %s\n", my_rank, count,
                   my_string);
            my_string = strtok(NULL, " \t\n");
        }
        sem_wait(&sems[my_rank]);
        fg_rv = fgets(my_line, MAX, stdin);
        sem_post(&sems[next]);
    }
    return NULL;
}
```

Running con un thread:

```
Pease porridge hot.  
Pease porridge cold.  
Pease porridge in the pot  
Nine days old.
```

Running con più threads:

```
Thread 0 > my line = Pease porridge hot.  
Thread 0 > string 1 = Pease  
Thread 0 > string 2 = porridge  
Thread 0 > string 3 = hot.  
Thread 1 > my line = Pease porridge cold.  
Thread 1 > string 1 = Pease  
Thread 1 > string 2 = porridge  
Thread 1 > string 3 = cold.  
Thread 2 > my line = Pease porridge in the pot  
Thread 2 > string 1 = Pease  
Thread 2 > string 2 = porridge  
Thread 2 > string 3 = in  
Thread 2 > string 4 = the  
Thread 2 > string 5 = pot  
Thread 3 > my line = Nine days old.  
Thread 3 > string 1 = Nine  
Thread 3 > string 2 = days  
Thread 3 > string 3 = old.
```

Strtok memorizza nella cache il puntatore alla riga di input dichiarando che una variabile ha una classe di archiviazione statica. Ciò fa sì che il valore memorizzato in questa variabile persista da una chiamata all'altra. Sfortunatamente per noi, questa stringa memorizzata nella cache è condivisa, non privata. Pertanto, la chiamata del thread 0 a strtok con la terza riga dell'input ha apparentemente sovrascritto il contenuto della chiamata del thread 1 con la seconda riga. Quindi la funzione strtok non è sicura per i thread. Se più thread lo chiamano contemporaneamente, l'output potrebbe non essere corretto.

In alcuni casi, lo standard C specifica una versione alternativa, sicura per i thread, di una funzione. In linea di principio “re-entrant”! = “thread safe”. In pratica, le funzioni re-entrant sono spesso anche thread safe. In caso di dubbio, controlla la documentazione! – (man strtok_r).

```
#include <string.h>  
char *strtok_r(char *str, const char *delim, char **saveptr);
```

Miscellaneous

Miscellaneous

Puoi usare le seguenti funzioni invece di pthread_mutex_init

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;  
pthread_cond_t cond_var = PTHREAD_COND_INITIALIZER;
```

Thread pinning

```
#define _GNU_SOURCE
#include <pthread.h>
void* thread_func(void* thread_args){
    ...
    cpu_set_t cpuset;
    pthread_t thread = pthread_self();
    /* Set affinity mask to include core 3 */
    CPU_ZERO(&cpuset);
    CPU_SET(3, &cpuset);
    s = pthread_setaffinity_np(thread, sizeof(cpu_set), &cpuset);
    ...
}
```

Timing code

```
#include <sys/time.h>
#define GET_TIME(now) {
    struct timeval t;
    gettimeofday(&t, NULL);
    now = t.tv_sec + t.tv_usec/1000000.0;
}
int main(){
    ...
    double start, finish, elapsed;
    GET_TIME(start);
    ...
    // Code to be timed
    ...
    GET_TIME(finish);
    elapsed = finish - start;
    printf("The code to be timed took %e seconds\n", elapsed);
    ...
}
```

Conclusioni

Un thread nella programmazione a memoria condivisa è analogo a un processo in programmazione a memoria distribuita.

In programmi Pthread, tutti i thread hanno accesso alle variabili globali, invece le variabili locali sono private per il thread che sta lavorando su quella funzione.

Quando l'indeterminazione deriva da più thread che tentano di accedere a una risorsa condivisa come una variabile condivisa o un file condiviso, almeno uno degli accessi è un aggiornamento, gli accessi possono causare un errore, abbiamo una **race condition**.

Una sezione critica è un blocco di codice che aggiorna una risorsa condivisa che può

essere aggiornata solo da un thread alla volta. Quindi l'esecuzione del codice in una sezione critica dovrebbe, di fatto, essere eseguita come codice seriale.

Busy-waiting può essere utilizzato per evitare l'accesso in conflitto alle sezioni critiche con una variabile flag e un ciclo while con un corpo vuoto. Può essere molto dispendioso di cicli della CPU. Può anche essere inaffidabile se l'ottimizzazione del compilatore è attivata.

Un mutex può essere utilizzato anche per evitare l'accesso in conflitto a sezioni critiche. Pensalo come un blocco su una sezione critica, poiché i mutex organizzano l'accesso reciprocamente esclusivo a una sezione critica.

Un semaforo è il terzo modo per evitare l'accesso conflittuale alle sezioni critiche. È un unsigned int insieme a due operazioni: sem_wait e sem_post. I semafori sono più potenti dei mutex poiché possono essere inizializzati a qualsiasi valore non negativo.

Una barriera è un punto in un programma in cui i thread si bloccano fino a quando tutti i thread non lo hanno raggiunto. Un blocco di lettura-scrittura viene utilizzato quando è sicuro per più thread leggere contemporaneamente una struttura dati, ma se un thread deve modificare o scrivere sulla struttura dati, solo quel thread può accedere alla struttura dati durante la modifica.

Alcune funzioni C memorizzano nella cache i dati tra le chiamate dichiarando che le variabili sono statiche, causando errori quando più thread chiamano la funzione. Questo tipo di funzione non è thread-safe.

3.7 Thread-safety in MPI

Invece che usare la funzione MPI_Init, usare la funzione:

```
int MPI_Init_thread(int *argc, char ***argv,
                    int required, /* in */ <<Required threading level>>
                    int *provided) /* out */ <<Supported threading level>>
```

Threading-levels in MPI

- MPI_THREAD_SINGLE : il rank non è autorizzato a utilizzare thread, il che è fondamentalmente equivalente a chiamare MPI_Init.
- MPI_THREAD_FUNNELED : il rank può essere multi-thread, ma solo il thread principale può chiamare le funzioni MPI. Ideale per il parallelismo fork-join come quello utilizzato in #pragma omp parallel, dove tutte le chiamate MPI sono al di fuori delle regioni OpenMP.
- MPI_THREAD_SERIALIZED : il rank può essere multi-thread, ma solo un thread alla volta può chiamare le funzioni MPI. Il rank deve garantire che MPI venga utilizzato in modo sicuro per i thread. Un approccio è garantire che l'utilizzo di MPI sia reciprocamente escluso da tutti i thread, ad esempio con un mutex.
- MPI_THREAD_MULTIPLE : il rank può essere multi-thread e qualsiasi thread può chiamare funzioni MPI. La libreria MPI garantisce che questo accesso sia sicuro su tutti i thread. Si noti che questo rende tutte le operazioni MPI meno efficienti, anche se solo un thread effettua chiamate MPI, quindi dovrebbe essere utilizzato solo dove necessario.

ATTENZIONE

Non tutti i livelli di threading sono supportati da tutte le implementazioni MPI (ad esempio, alcune implementazioni potrebbero non supportare MPI_THREAD_MULTIPLE)

3.8 Caching

Una raccolta di posizioni di memoria a cui è possibile accedere in meno tempo rispetto ad altre posizioni di memoria. Una cache della CPU si trova in genere sullo stesso chip o a cui è possibile accedere molto più velocemente della memoria ordinaria (di solito è fisicamente più vicina).

Utilizza anche una tecnologia più performante (ma anche più costosa) (ad esempio, SRAM invece di DRAM). Quindi sarà più veloce ma più piccola.

Cosa memorizzare nella cache

Assumiamo la località (sia per le istruzioni che per i dati). Ad esempio, l'accesso a una posizione è seguito da un accesso a una posizione vicina.

Località spaziale – accesso a una posizione vicina.

Località temporale – accedervi nel prossimo futuro.

Linee di cache

I dati vengono trasferiti dalla memoria alla cache in blocchi/linee, cioè, quando $z[0]$ viene trasferito dalla memoria alla cache, anche $z[1]$, $z[2]$, ..., $z[15]$ potrebbe essere trasferito. Fare un trasferimento di 16 posizioni di memoria, è meglio che fare 16 trasferimenti di una posizione di memoria ciascuno. Quando si accede a $z[0]$ devi aspettare il trasferimento, ma poi troverai già gli altri 15 elementi nella cache.

Perché ci interessa?

Per scrivere codice parallelo efficiente/performante:

- Le sue parti sequenziali devono essere efficienti/performanti (Prova a pensare a come la tua applicazione accede ai dati.)
- Gli accessi casuali sono molto peggiori degli accessi lineari
- Il coordinamento tra queste parti sequenziali deve essere fatto in modo efficiente

Coerenza

Quando una CPU scrive dati nella cache, il valore nella cache potrebbe essere incoerente con il valore nella memoria principale. Le cache write-through gestiscono questo aggiornando i dati nella memoria principale nel momento in cui vengono scritti nella cache. Le cache di write-back contrassegnano i dati nella cache come sporchi. Quando la linea di cache viene sostituita da una nuova riga di cache dalla memoria, la linea sporca viene scritta in memoria.

Coerenza della cache

I programmati non hanno alcun controllo sulle cache e su quando vengono aggiornate.

Snooping Cache Coherence

I core condividono un bus. Qualsiasi segnale trasmesso sul bus può essere "visto" da tutti i core collegati al bus. Quando il core 0 aggiorna la copia di x memorizzata nella sua cache, trasmette anche queste informazioni attraverso il bus. Se il core 1 sta "snoopando" il bus, vedrà che x è stato aggiornato e può contrassegnare la sua copia di x come non valida. Non è più molto usato: la trasmissione è costosa.

Coerenza della cache basata su directory

Utilizza una struttura dati chiamata directory che memorizza lo stato di ogni riga di cache.

Ad esempio, una bitmap/elenco che dice quali core hanno una copia di quella riga.

Quando una variabile viene aggiornata, la directory viene consultata e i controller di cache dei core che hanno la linea di cache di quella variabile nelle loro cache vengono invalidati.

False sharing

I dati vengono recuperati per la memorizzazione nella cache in righe. Ogni riga può contenere diverse variabili

Ad esempio, se una cache è lunga 64 byte, può contenere 16 numeri interi a 4 byte (che erano consecutivi in memoria)

Quando i dati vengono invalidati, l'intera riga viene invalidata. Anche se due thread accedono a due variabili diverse, se quelli sono sulla stessa riga della cache, ciò causerebbe comunque un'invalidazione.

Soluzione.

Prova a forzare le variabili a cui si accede da thread diversi a essere su diverse linee di cache Padding (ma attenzione, il compilatore potrebbe cambiare l'ordine dei campi in una struttura). Per evitarlo, fai qualcosa del genere (per GCC, assumendo linee di cache di 64 byte):

```
struct alignTo64ByteCacheLine{  
    int _onCacheLine1 __attribute__((aligned(64)))  
    int _onCacheLine2 __attribute__((aligned(64)))  
}
```

Come ottenere la dimensione della linea di cache?

Dal codice:

```
sysconf (_SC_LEVEL1_DCACHE_LINESIZE)
```

Dalla shell:

```
getconf LEVEL1_DCACHE_LINESIZE
```

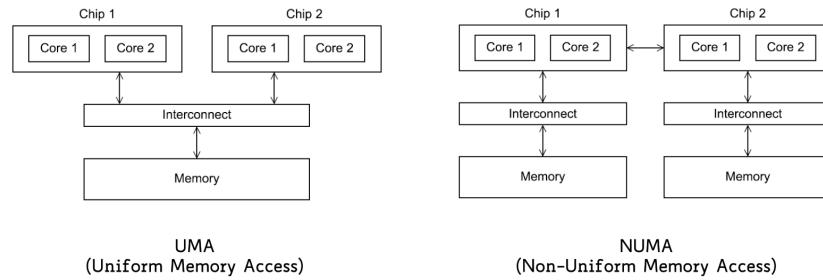
```
perf stat -e cache-misses ./cache_fs_slow  
perf stat -e cache-misses ./cache_fs_fast
```

Alternatively, if supported:

```
perf c2c record ./cache_fs_slow  
perf c2c report
```

Figure 31: Come vedere le prestazioni di un programma

Memory Organization



NUMA System

Il costo di accesso alla memoria cambia a seconda di dove sono assegnati i dati. Accedere a una memoria "locale" è più economico rispetto all'accesso a una memoria "remota". È possibile specificare dove devono essere assegnati i dati (vedi libreria numa.h). È possibile avere un certo controllo dalla riga di comando (vedi numactl).

4 OpenMP