Programmazione Concorrente e Parallela

Jacopo Castellini

A.A. 2015/16

Consegna

Il progetto da svolgere consiste nella parallelizzazione della seguente procedura per calcolare la decomposizione LU di una matrice diagonale a bande:

```
#include <math.h>
#define SWAP(a,b) {dum=(a);(a)=(b);(b)=dum;}
#define TINY 1.0e-20

void bandec(float **a, unsigned long n, int m1, int m2, float **al,
	unsigned long indx[], float *d)
{
	unsigned long i,j,k,l;
	int mm;
	float dum;

	mm=m1+m2+1;
	l=m1;
	for (i=1;i<=m1;i++) {
		for (j=m1+2-i;j<=mm;j++) a[i][j-l]=a[i][j];
		l--;
		for (j=mm-l;j<=mm;j++) a[i][j]=0.0;
	}
	*d=1.0;
	l=m1;
	for (k=1;k<=n;k++) {
		dum=a[k][1];
		i=k;
		if (l < n) l++;
		for (j=k+1;j<=l;j++) {
			if (fabs(a[j][1]) > fabs(dum)) {
				dum=a[j][1];
				i=j;
			}
		}
		indx[k]=i;
		if (dum == 0.0) a[k][1]=TINY;
		if (i != k) {
			*d = -(*d);
			for (j=1;j<=mm;j++) SWAP(a[k][j],a[i][j])
		}
		for (i=k+1;i<=l;i++) {
			dum=a[i][1]/a[k][1];
			al[k][i-k]=dum;
			for (j=2;j<=mm;j++) a[i][j-1]=a[i][j]-dum*a[k][j];
			a[i][mm]=0.0;
		}
	}
}
#undef SWAP
#undef TINY
/* (C) Copr. 1986-92 Numerical Recipes Software ?421.1-9. */
```

Parallelizzazione

Per parallelizzare l'algoritmo abbiamo scelto un approccio master-workers, in cui il processo con rank = 0 è il master (colui che distribuisce i lavori e raccoglie i risultati, ma che non esegue calcoli), mentre tutti gli altri sono workers (cioè aspettano richieste dal master ed elaborano i dati da esso ricevuti, poi gli restituiscono i risultati). Inoltre i lavori vengono distribuiti dinamicamente, in modo da ridurre al minimo il tempo di attesa per il completamento dei lavori, e le comunicazioni sono svolte in modo asincrono dove possibile (cioè esclusivamente nelle send, poiché i valori ricevuti nelle receive sono sempre utilizzati immediatamente dopo averli ricevuti). Ci è sembrato possibile parallelizzare il codice originale esclusivamente in due punti (cosa tra l'altro messa in evidenza dagli stessi autori del libro in Numical Recipes in Fortran 90, Volume 2) parallelizzando il lavoro svolto sulle colonne della matrice (cicli for più interni) invece che per riga, poiché ogni operazione fatta su una riga (ciclo for più esterno) richiede il completamento di quella sulla riga precedente, e risulta quindi non parallelizzabile.

Il primo punto che è stato parallelizzato è il ciclo in cui vengono shiftati i valori delle prime righe della matrice:

```
for (i=1;i<=m1;i++) {
        for (j=m1+2-i;j<=mm;j++) a[i][j-l]=a[i][j];
        l--;
        for (j=mm-l;j<=mm;j++) a[i][j]=0.0;
}
```

che è stato parallelizzato distribuendo le righe tra i vari processi (qui è possibile lavorare sulle righe perché ogni operazione è locale alla riga stessa). Il processo master eseguirà questo codice:

```
l = m1;
if(min > m1)
        min = m1;
MPI_Request req_list[world_size - 1];
for(i = 1; i < min + 1; i++)
        MPI_Isend(&a[i - 1][0], 1, rowtype, i, i - 1, MPI_COMM_WORLD, &req_list[i - 1]);
for(i = min + 1; i < world_size; i++)
        MPI_Isend(&a[0][0], 1, rowtype, i, -1, MPI_COMM_WORLD, &req_list[i - 1]);
for(i = min; i < m1 + min; i++) {
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        sender = status.MPI_SOURCE;
        tag = status.MPI_TAG;
        MPI_Recv(&a[tag][0], 1, rowtype, sender, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if(i < m1)
                MPI_Isend(&a[i][0], 1, rowtype, sender, i, MPI_COMM_WORLD, &req_list[sender]);
        else
                MPI_Isend(&a[0][0], 1, rowtype, sender, -1, MPI_COMM_WORLD, &req_list[sender]);
}
```

Questo innanzi tutto controlla se sono di più i processi workers o le righe da elaborare, poiché se ci sono più processi che lavoro il master deve inviare a quelli rimasti disoccupati un messaggio che gli indichi che possono proseguire nella loro esecuzione (inoltre, poiché in MPI ogni processore richiesto deve tassativamente svolgere del lavoro, questo è un buon modo di coinvolgerli tutti). I processi workers invece eseguiranno il codice:

```
while(1) {
        MPI_Recv(&temp[0], 1, rowtype, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        tag = status.MPI_TAG;
        if(tag == -1)
                break;
        l = m1 - tag;
```

```
        for(j = m1 - tag; j < mm; j++)
                temp[j - l] = temp[j];
        l--;
        for(j = mm - l - 1; j < mm; j++)
                temp[j] = 0.0;
        MPI_Send(&temp[0], 1, rowtype, 0, tag, MPI_COMM_WORLD);
}
```

che altro non è che il reale lavoro di calcolo dell'algoritmo.

Il secondo punto invece è l'ultimo ciclo for interno all'algoritmo di decomposizione vero e proprio, in cui vengono calcolati i valori delle matrici L e U (rispettivamente al e a nel codice):

```
for (i=k+1;i<=l;i++) {
        dum=a[i][1]/a[k][1];
        al[k][i-k]=dum;
        for (j=2;j<=mm;j++) a[i][j-1]=a[i][j]-dum*a[k][j];
        a[i][mm]=0.0;
}
```

Il processo master eseguirà il codice:

```
min = world_size - 1;
if(min > l - k - 1)
        min = l - k - 1;
MPI_Waitall(world_size - 1, req_list, MPI_STATUSES_IGNORE);
for(i = 1; i < min + 1; i++) {
        dum = a[i + k][0] / a[k][0];
        al[k][i - 1] = dum;
        MPI_Isend(&a[k][0], 1, rowtype, i, 0, MPI_COMM_WORLD, &req_list[i - 1]);
        MPI_Isend(&a[i + k][0], 1, rowtype, i, i + k, MPI_COMM_WORLD, &req_list[i - 1]);
}
for(i = min + 1; i < world_size; i++)
        req_list[i - 1] = MPI_REQUEST_NULL;
for(i = min + 1; i < l - k + min; i++) {
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        sender = status.MPI_SOURCE;
        tag = status.MPI_TAG;
        MPI_Recv(&a[tag][0], 1, rowtype, sender, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if(i < l - k) {
                dum = a[i + k][0] / a[k][0];
                al[k][i - 1] = dum;
                MPI_Isend(&a[i + k][0], 1, rowtype, sender, i + k, MPI_COMM_WORLD, &req_list[sender]);
        } else
                MPI_Isend(&a[0][0], 1, rowtype, sender, -1, MPI_COMM_WORLD, &req_list[sender]);
}
```

e, quando il ciclo for più esterno (quello che itera su k) è terminato, viene notificata la fine dei lavori ai processi workers con il codice:

```
for(i = 1; i < world_size; i++)
        MPI_Isend(&a[0][0], 1, rowtype, i, -2, MPI_COMM_WORLD, &req_list[i – 1]);
```

Oltre ad un discorso analogo al primo ciclo parallelizzato per quanto riguarda il numero di lavori e di workers, il processo master aggiorna da sé la matrice al, poiché non si avrebbero vantaggi (ma solo costi in più dovuti alle comunicazioni) nel delegare anche questo compito ai processi workers. Questi ultimi invece eseguiranno il codice:

```
while(1) {
	MPI_Recv(&act[0], 1, rowtype, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
	tag = status.MPI_TAG;
	if(tag == -2)
		break;
	while(tag != -1) {
		MPI_Recv(&temp[0], 1, rowtype, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
		tag = status.MPI_TAG;
		if(tag == -1)
			break;
		dum = temp[0] / act[0];
		for(j = 1; j < mm; j++)
			temp[j - 1] = temp[j] - dum * act[j];
		temp[mm - 1] = 0.0;
		MPI_Send(&temp[0], 1, rowtype, 0, tag, MPI_COMM_WORLD);
	}
}
```

in cui eseguono il lavoro vero e proprio dell'algoritmo, oltre ad effettuare dei controlli sull'eventuale fine dei lavori. Il resto dell'algoritmo è stato lasciato seriale (e quindi svolto dal master). Di seguito l'intero codice parallelo dell'algoritmo:

```
#include <stdio.h>
#include <math.h>
#include </opt/lib/mpi/intel/14.0.2/mvapich2/1.9/include/mpi.h>
#define N 5000
#define m1 800
#define m2 400
#define mm (m1 + m2 + 1)
#define SWAP(c,b) {dum=(c);(c)=(b);(b)=dum;}
#define TINY 1.0e-20

void create_matrix(double ***a, double ***al) {
	(*a) = (double **) malloc(N * sizeof(double *));
	(*al) = (double **) malloc(N * sizeof(double *));
	int i, j;
	for(i = 0; i < N; i++) {
		(*al)[i] = (double *) malloc(m1 * sizeof(double));
		(*a)[i] = (double *) malloc(mm * sizeof(double));
	}
	for(i = 0; i < N; i++)
		for(j = 0; j < m1; j++)
			(*al)[i][j] = 0.0;
	for(i = 0; i < m1; i++) {
		for(j = 0; j < m1 - i; j++)
			(*a)[i][j] = 0.0;
		for(j = m1 - i; j < mm; j++)
			(*a)[i][j] = (double) (j + 1);
	}
	for(i = m1; i < N - m2; i++)
		for(j = 0; j < mm; j++)
			(*a)[i][j] = (double) (j + 1);
	for(i = N - m2; i < N; i++) {
		for(j = 0; j < mm - (N - i); j++)
			(*a)[i][j] = (double) (j + 1);
		for(j = mm - (N - i); j < mm; j++)
			(*a)[i][j] = 0.0;
	}
}

int main(int argc, char **argv) {
	MPI_Init(&argc, &argv);
```

```
	double s_time = MPI_Wtime();
	int world_size, my_rank, d = 1, i, j, k, l, tag;
	double dum;
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);
	MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
	MPI_Datatype rowtype;
	MPI_Status status;
	MPI_Type_contiguous(mm, MPI_DOUBLE, &rowtype);
	MPI_Type_commit(&rowtype);
	if(my_rank == 0) {
		int min = world_size - 1, sender, indx[N];
		double **a, **al;
		create_matrix(&a, &al);
		l = m1;
		if(min > m1)
			min = m1;
		MPI_Request req_list[world_size - 1];
		for(i = 1; i < min + 1; i++)
			MPI_Isend(&a[i - 1][0], 1, rowtype, i, i - 1, MPI_COMM_WORLD, &req_list[i - 1]);
		for(i = min + 1; i < world_size; i++)
			MPI_Isend(&a[0][0], 1, rowtype, i, -1, MPI_COMM_WORLD, &req_list[i - 1]);
		for(i = min; i < m1 + min; i++) {
			MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
			sender = status.MPI_SOURCE;
			tag = status.MPI_TAG;
			MPI_Recv(&a[tag][0], 1, rowtype, sender, tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
			if(i < m1)
				MPI_Isend(&a[i][0], 1, rowtype, sender, i, MPI_COMM_WORLD,
&req_list[sender]);
			else
				MPI_Isend(&a[0][0], 1, rowtype, sender, -1, MPI_COMM_WORLD,
&req_list[sender]);
		}
		l = m1;
		for(k = 0; k < N; k++) {
			dum = a[k][0];
			i = k;
			if(l < N)
				l++;
			for(j = k + 1; j < l; j++) {
				if(fabs(a[j][0]) > fabs(dum)) {
					dum = a[j][0];
					i = j;
				}
			}
			indx[k] = i;
			if(dum = 0.0)
				a[k][0] = TINY;
			if(i != k) {
				d = -d;
				for(j = 0; j < mm; j++)
					SWAP(a[k][j], a[i][j]);
			}
			min = world_size - 1;
			if(min > l - k - 1)
				min = l - k - 1;
			MPI_Waitall(world_size - 1, req_list, MPI_STATUSES_IGNORE);
			for(i = 1; i < min + 1; i++) {
				dum = a[i + k][0] / a[k][0];
				al[k][i - 1] = dum;
				MPI_Isend(&a[k][0], 1, rowtype, i, 0, MPI_COMM_WORLD, &req_list[i - 1]);
```

```
                                        MPI_Isend(&a[i + k][0], 1, rowtype, i, i + k, MPI_COMM_WORLD, &req_list[i -
1]);
                                }
                                for(i = min + 1; i < world_size; i++)
                                        req_list[i - 1] = MPI_REQUEST_NULL;
                                for(i = min + 1; i < l - k + min; i++) {
                                        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
                                        sender = status.MPI_SOURCE;
                                        tag = status.MPI_TAG;
                                        MPI_Recv(&a[tag][0], 1, rowtype, sender, tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                                        if(i < l - k) {
                                                dum = a[i + k][0] / a[k][0];
                                                al[k][i - 1] = dum;
                                                MPI_Isend(&a[i + k][0], 1, rowtype, sender, i + k, MPI_COMM_WORLD,
&req_list[sender]);
                                        } else
                                                MPI_Isend(&a[0][0], 1, rowtype, sender, -1, MPI_COMM_WORLD,
&req_list[sender]);
                                }
                        }
                        for(i = 1; i < world_size; i++)
                                MPI_Isend(&a[0][0], 1, rowtype, i, -2, MPI_COMM_WORLD, &req_list[i - 1]);
                } else {
                        double temp[mm], act[mm];
                        while(1) {
                                MPI_Recv(&temp[0], 1, rowtype, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                                tag = status.MPI_TAG;
                                if(tag == -1)
                                        break;
                                l = m1 - tag;
                                for(j = m1 - tag; j < mm; j++)
                                        temp[j - l] = temp[j];
                                l--;
                                for(j = mm - l - 1; j < mm; j++)
                                        temp[j] = 0.0;
                                MPI_Send(&temp[0], 1, rowtype, 0, tag, MPI_COMM_WORLD);
                        }
                        while(1) {
                                MPI_Recv(&act[0], 1, rowtype, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                                tag = status.MPI_TAG;
                                if(tag == -2)
                                        break;
                                while(tag != -1) {
                                        MPI_Recv(&temp[0], 1, rowtype, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
                                        tag = status.MPI_TAG;
                                        if(tag == -1)
                                                break;
                                        dum = temp[0] / act[0];
                                        for(j = 1; j < mm; j++)
                                                temp[j - 1] = temp[j] - dum * act[j];
                                        temp[mm - 1] = 0.0;
                                        MPI_Send(&temp[0], 1, rowtype, 0, tag, MPI_COMM_WORLD);
                                }
                        }
                }
                MPI_Type_free(&rowtype);
                printf("Time for process %d: %f\n", my_rank, MPI_Wtime() - s_time);
                MPI_Finalize();
}
#undef SWAP
```

#undef TINY

Risultati

Vediamo ora alcuni esperimenti per analizzare i risultati dell'approccio parallelo utilizzato. Abbiamo scelto di utilizzare il compilatore della Intel e la distribuzione di MPI chiamata mvapich2. Andremo ad analizzare i risultati in due modi: in termini di SpeedUp, cioè di quanto l'implementazione parallela risulti più veloce di quella seriale, ed in termini di Efficienza, cioè di quanto bene vengano sfruttate le risorse a disposizione del programma.

Possiamo definire lo SpeedUp *S* per una certa implementazione parallela come:

$$S = \text{tempo seriale / tempo parallelo,}$$

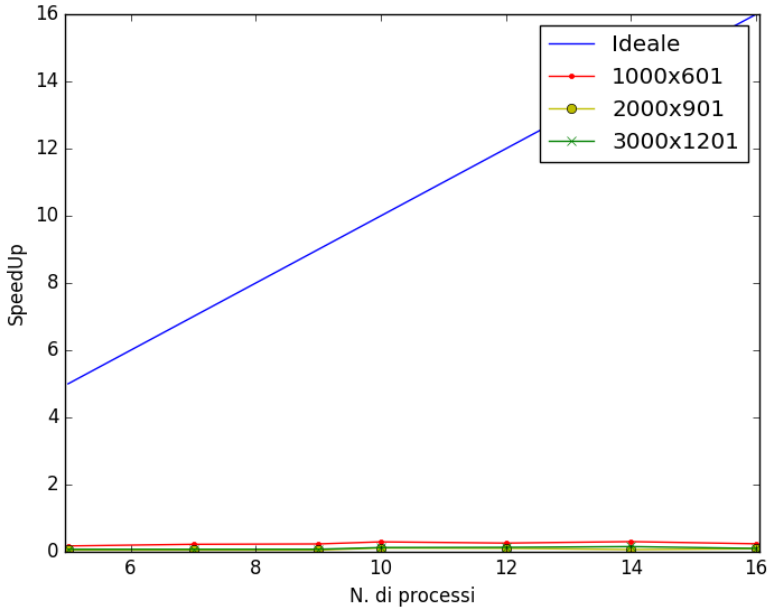
mentre l'Efficienza *E* sarà:

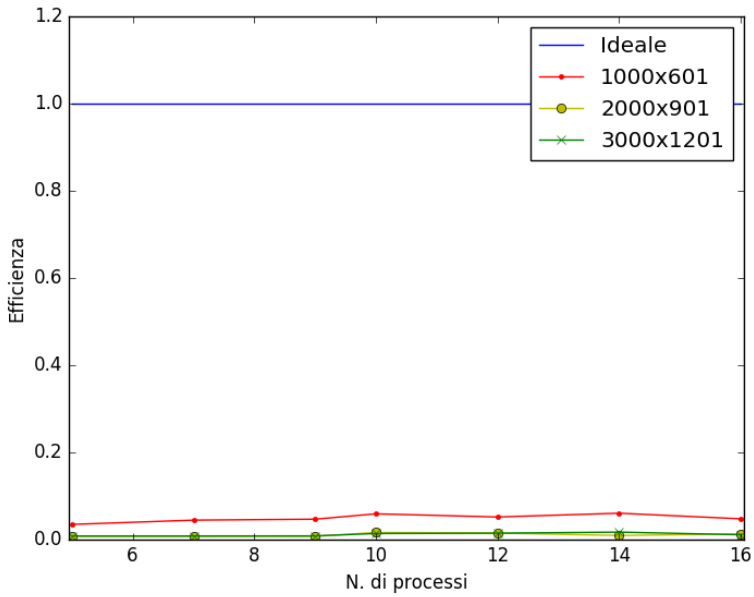
$$E = S \text{ / n. processi.}$$

Useremo matrici di varie dimensioni e con un numero variabile di diagonali non nulle, così da poter meglio analizzare quando l'implementazione parallela sia più o meno efficiente.

Dimensioni	Seriale	5 proc.	7 proc.	9 proc.	10 proc.	12 proc.	14 proc.	16 proc.
1000x601	3.00	17.20	13.44	12.81	10.15	11.57	9.91	12.59
2000x901	10.00	192.58	193.39	191.76	87.57	95.06	151.33	109.71
3000x1201	27.00	361.43	361.95	355.12	214.34	204.48	174.40	270.07

Appare evidente come un implementazione parallela di questo tipo non è affatto efficace per risolvere il problema, in quanto l'eccessivo numero di comunicazioni necessarie annulla il vantaggio così ottenuto, e anzi aumenta vertiginosamente il tempo di esecuzione. I grafici dello SpeedUp e dell'Efficienza sono una conferma di questo fatto:





Poiché il parallelismo è svolto sulle colonne, il numero di iterazioni del for più esterno (quello che scandisce le righe) eseguito dal processo master è uguale a quello della versione seriale. Inoltre il costo delle comunicazioni è più alto di quello che si avrebbe eseguendo i calcoli in maniera seriale. Tutto ciò porta ad una generale inefficacia dell'approccio al problema trattato. In conclusione, la fattorizzazione LU con pivoting di una matrice a bande diagonali sembra essere un problema che non si adatta bene ad essere parallelizzato, neanche quando le dimensioni in gioco aumentano. Ciò è dovuto ad un'interdipendenza tra le iterazioni sulle righe della matrice che impediscono un pieno parallelismo, ostacolo non superabile neanche con l'utilizzo della memoria condivisa.