Fattorizzazione LU di matrici diagonali a bande

Jacopo Castellini
A.A. 2015/16

Il problema

Il nostro scopo è quello di fornire un implementazione parallela dell'algoritmo per la decomposizione LU di matrici diagonali a bande utilizzando MPI.

La decomposizione LU (con pivoting) di una matrice A costruisce due matrici, L triangolare inferiore e U triangolare superiore, tali che LU = (una permutazione di) A.

Il problema

Una matrice a bande diagonali è una matrice di dimensione NxN che ha tutti gli elementi al di fuori della diagonale, delle prime m1 sottodiagonali e delle prime m2 sopradiagonali nulli.

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 0 & 0 & 0 \\ 9 & 2 & 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 9 & 0 & 0 \\ 0 & 0 & 7 & 9 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 8 & 4 & 6 \\ 0 & 0 & 0 & 0 & 2 & 4 & 4 \end{pmatrix}$$

Il problema

Per risparmiare memoria ed aumentare l'efficienza degli algoritmi utilizzati, una matrice a bande può essere memorizzata nella cosiddetta forma compatta, di dimensioni Nx(m1 + m2 + 1).

$$\begin{pmatrix} x & x & 3 & 1 \\ x & 4 & 1 & 5 \\ 9 & 2 & 6 & 5 \\ 3 & 5 & 8 & 9 \\ 7 & 9 & 3 & 2 \\ 3 & 8 & 4 & 6 \\ 2 & 4 & 4 & x \end{pmatrix}$$

La parallelizzazione

Abbiamo scelto un approccio master-workers ed un bilanciamento del carico di tipo dinamico. Inoltre si è cercato di utilizzare chiamate asincrone ove possibile, in modo da ridurre al minimo i tempi di attesa del programma.

La parallelizzazione

Abbiamo parallelizzato l'algoritmo originale in due punti: il primo è lo shift a sinistra dei valori sulle prime righe della matrice (primo ciclo for nel codice originale):

```
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;
}
```

Abbiamo scelto di distribuire le righe tra i vari processi workers.

La parallelizzazione

Il processo master (rank = 0) eseguirà:

```
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]);
}
```

La parallelizzazione

Mentre i processi workers faranno i calcoli veri e propri:

```
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);
}
```

La parallelizzazione

Il secondo punto invece è quello in cui vengono aggiornati i valori della matrice U (a nel codice) e L (al nel codice), cioè l'ultimo for interno nel codice originale:

```
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;
}
```

Questa volta non è stato possibile distribuire il lavoro per righe (cioè parallelizzare il for più esterno di questo), quindi viene parallelizzato per colonne.

La parallelizzazione

Il processo master eseguirà:

```
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]);
}
```

La parallelizzazione

Mentre i workers faranno i conti (e controlleranno l'eventuale fine dei lavori):

```
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);
        }
}
```

Risultati

Vediamo ora se l'approccio parallelo utilizzato è risultato efficiente. Definiamo intanto due misure, lo speedup S e l'efficienza E:

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

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

Risultati

Abbiamo eseguito il codice variando il numero di processi utilizzati e la dimensione della matrice (sia il numero di righe che il numero di diagonali non nulle). Ecco i risultati:

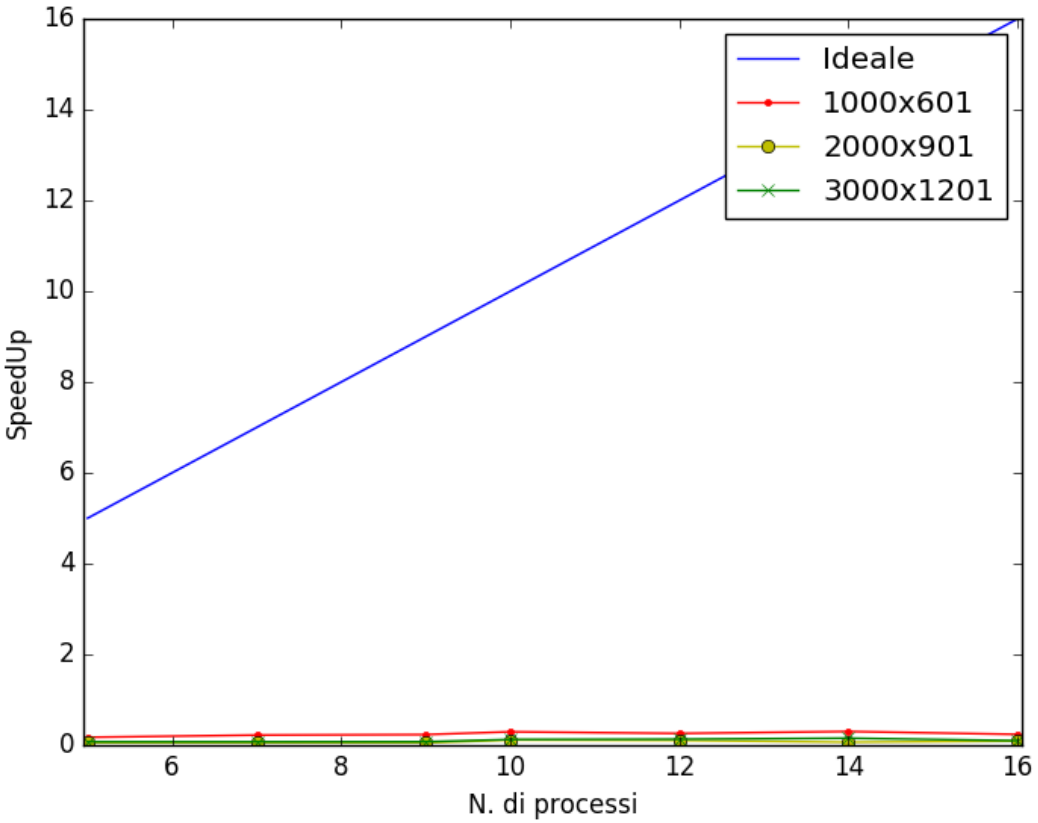
Dimensioni	Seriale	5 proc.	7 proc.	9 proc.	10 proc.	12 proc.	14 proc.	16 proc.
1000x601	3.00	17.20	13.44	12.81	10.15	11.57	9.91	12.59
2000x901	10.00	192.58	193.39	191.76	87.57	95.06	151.33	109.71
3000x1201	27.00	361.43	361.95	355.12	214.34	204.48	174.40	270.07

Risultati

Appare evidente come un approccio parallelo non risulti buono per il problema affrontato, in quanto l'eccessivo numero di comunicazioni utilizzate portano ad un degrado delle prestazioni invece che ad un risparmio di tempo.

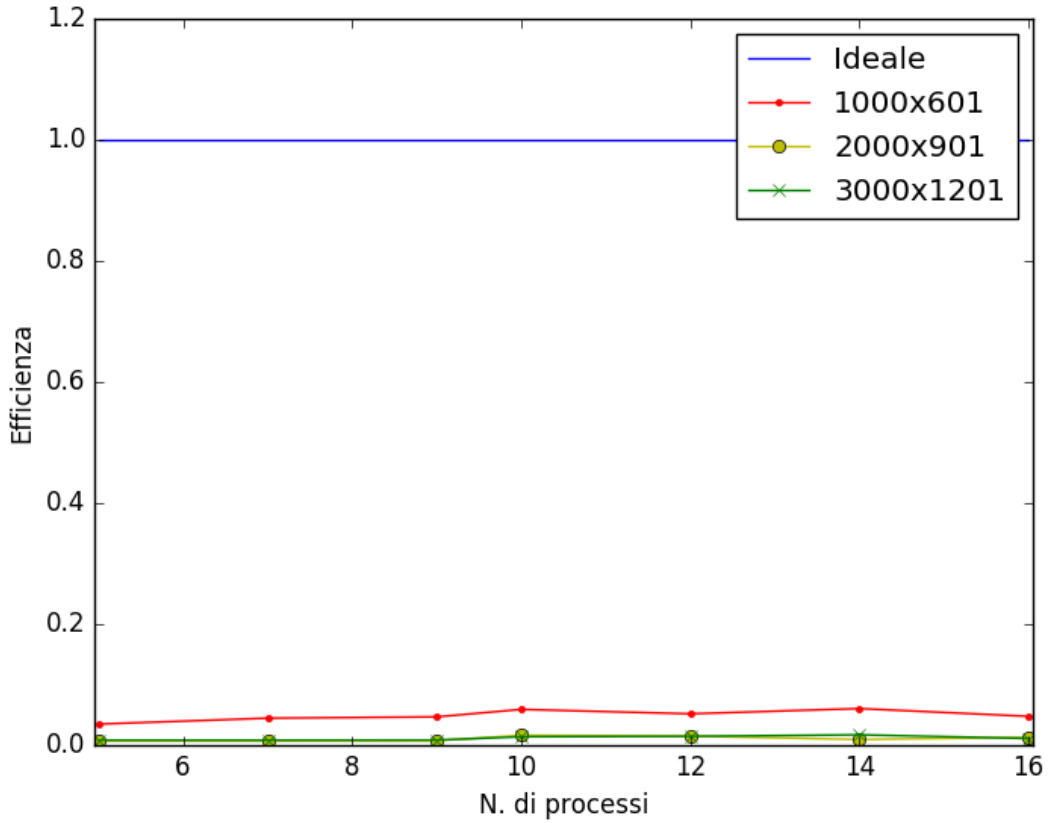
Risultati

Grafico dello speedup S



Risultati

Grafico dell’efficienza E



Conclusioni

Appare evidente come questo approccio, cioè l'idea di distribuire il lavoro sulle colonne della matrice da decomporre, impedisca di ottenere risultati validi dalla parallelizzazione. Il meccanismo di pivoting usato per la stabilità numerica dell'algoritmo impedisce una parallelizzazione sulle righe, che invece avrebbe portato a risultati ben migliori, poiché crea una interdipendenza totale tra una iterazione e la sua precedente. Neanche l'uso di memoria condivisa avrebbe giovato al nostro approccio, proprio in virtù di ciò.