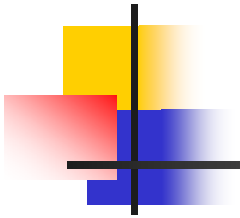


並列アルゴリズム

2005年後期　火曜　2限

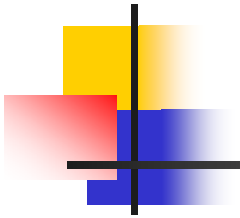
高見　利也（青柳　睦）

Aoyagi@cc.kyushu-u.ac.jp

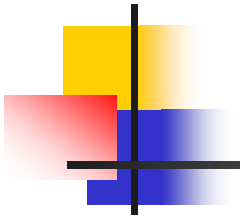
http://server-500.cc.kyushu-u.ac.jp/

12月20日（火）

9. PCクラスタによる並列プログラミング（演習） つづき



もくじ



今日の内容

- パフォーマンス測定の座標軸
  - 基礎知識
    - 計算量、浮動小数点演算回数
  - 11/08の講義の復習と実例
    - 問題サイズとCPUアーキテクチャ
    - 並列化による所要時間の短縮
    - Speed Up Ratio, Overhead
- 並列化の方針と予想
  - block分割／cyclic分割／block-cyclic分割
- 演習
  - 問題サイズ依存性、並列化の効果を確かめる。

パフォーマンス測定の予備知識

- 計算量のカウント
  - 浮動小数点演算の回数を数える
  - LU分解の場合：

$$\sum_{k=0}^{n-2}[n-1-k+2(n-1-k)^2]=\frac{2}{3}n^3+\ldots$$

```
for (k = 0; k < n-1; k++) {
    for (j = k+1; j < n; j++)
        a[k][j] /= a[k][k];
    for (i = k+1; i < n; i++)
        for ( j = k+1; j < n; j++)
            a[i][j] -= a[i][k] * a[k][j];
}
```

LU分解のcore部分

- 浮動小数点演算性能(FLoating-point number Operations Per Second)
  - Flops値 = 浮動小数点演算回数／計算時間(秒)
  - 実際の単位： MegaFlops (10^6flops), GigaFlops (10^9flops)
    他に Kilo = 10^3, Tera = 10^{12}, Peta = 10^{15} など
- 具体例
  - 1000x1000の行列を1GFlopsの実効性能でLU分解したとすると

$$\frac{2}{3}\times 1000^3 \Big/ 10^9 = 0.67 \quad (秒)$$

パフォーマンス測定の座標軸(1)

- **数値演算性能の問題サイズ依存性**： CPUアーキテクチャによる違い
  - 実効性能を1秒あたりの浮動小数点演算回数で測定するとき、問題サイズによって実効性能は大きく変わる。
  - それだけでなく、この性質は、計算機の種類でまったく異なるものとなる。

- **具体例(linpackc.c)**
  - PCの場合、キャッシュを利用できる範囲では高性能だが、サイズが大きいと、メモリバンド幅で決まる一定性能に落ち着く。
  - ベクトル型スパコンなどで、メモリバンド幅が非常に大きいものの場合は、サイズが大きくなればなるほど性能は伸びる。

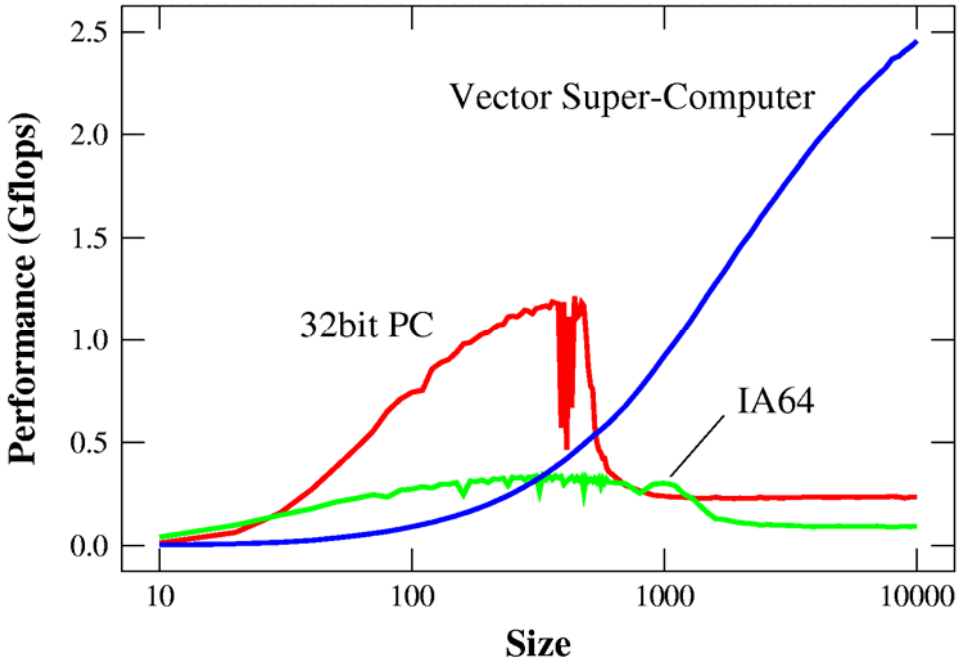


図1 アーキテクチャの異なるマシン性能のサイズ依存性比較

パフォーマンス測定の座標軸(2)

- **並列化による演算所要時間短縮**
  - 一般に並列化が有効な場合、CPU数が大きくなれば、それに応じて所要時間は短縮されるはず(Linear Speedup)。現実には(特に、問題サイズが小さい時)、さまざまな要因(Overheadなど)で、十分な効果が得られない場合がある。
  - 逆に、もともとの問題サイズが1 CPUで実行するには大きすぎるような場合には、CPU数以上の並列効果が見られる場合がある(Super-linear Speedup)。

- **具体例**
  - 右図は、IA64 SMP並列マシンのOpenMPによるLU分解の並列化効率である。
  - 小さいサイズの問題に対しては、あまり効果がないが、大きい問題に対しては、逆にCPU数以上の効果が見られる。
  - 横軸は利用CPU数、縦軸は 1 CPUの時を1とした場合の所要時間

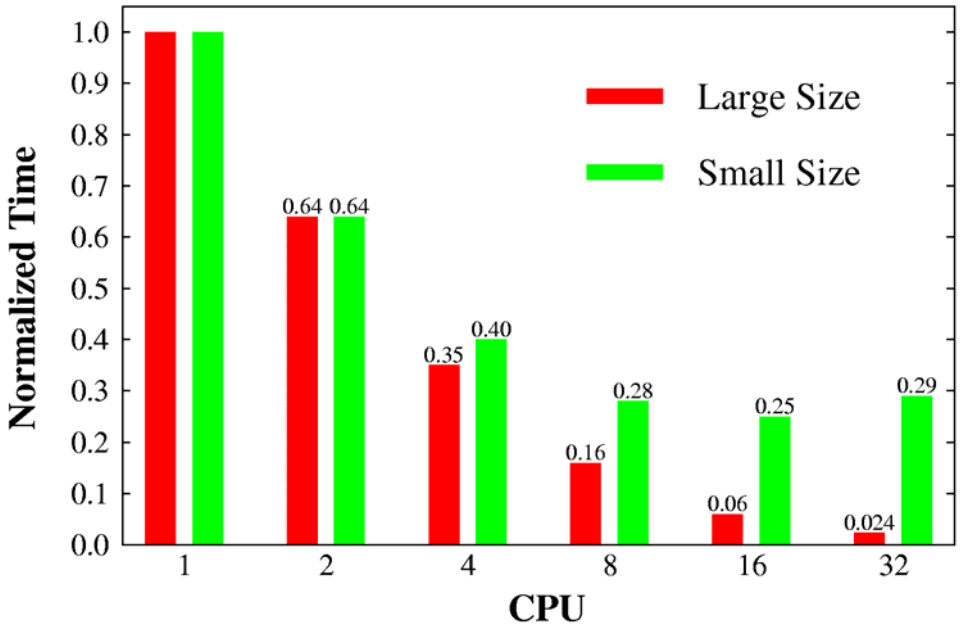


図2　並列化による時間短縮効果の問題サイズ依存性

パフォーマンス測定の座標軸(3)

- **並列化によるスピードアップ**(Speedup Ratio)
  - 横軸にCPU数をとり、縦軸に演算性能をプロットすると、理想的な場合には直線が描かれる(linear speedup) 。
  - しかし、実際には、その右側に来ることが多い。
    - Overheadや通信時間の影響
  - 場合によっては、直線よりも上回ることもある(super-linear speedup)。
    - 1 CPUの場合の性能が悪すぎる。
    - 一般にCPU毎にキャッシュなどで決まる最適問題サイズが存在する。

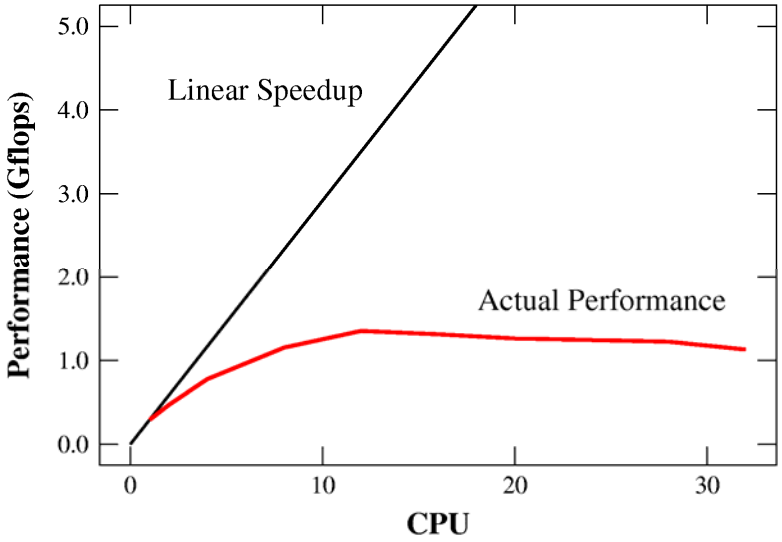


図3　実際はlinear speedupにはならない

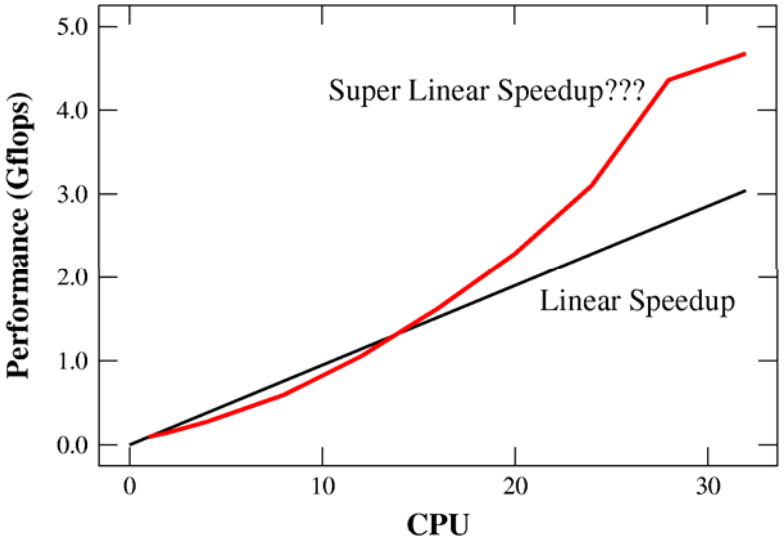
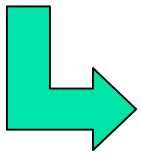


図4　Super-linear speedupの例

LU分解の逐次プログラムを並列化する

```
void
lu_s(int n, mat_t a)
{
  int i,j,k;

  for(k=0;k<n-1;k++)
    {
        for(j=k+1;j<n;j++)
            a[k][j] /= a[k][k];

        for(i=k+1;i<n;i++)
          for(j=k+1;j<n;j++)
            a[i][j] -= a[i][k] * a[k][j];
    }
}
```

```
        :
/* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

/* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        :
   genmat(n, a );
        :
   lu_p(n, a);
        :
```



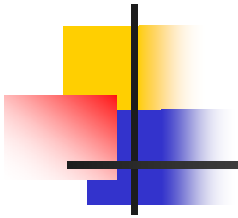
並列化

- 行についてcyclic またはblock分割を試みる
- 自Rankが持っていない要素の位置に注意
- 処理が終わったらRank0に集約する

LU分解アルゴリズムの並列化例(R-L法）

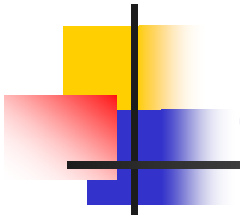
```
void  lu_p(int n, mat_t a)
{
 int i,j,k;
 int MAP[SIZE];
 int blk;

 /* cyclic decomposition */
 for(i=0;i<n;i++) MAP[i]= i % numprocs;

 /* block decomposition
 blk=n/numprocs;
 for(i=0;i<numprocs;i++)
   for(j=0;j<blk;j++) MAP[i*blk + j]= i; */

for(k=0;k<n-1;k++)
  {
     if( MAP[k] == my_rank )
        for(j=k+1;j<n;j++)
          a[k][j] /= a[k][k];

  MPI_Bcast(&a[k][k+1],n-k-1,MPI_DOUBLE,
                  MAP[k],MPI_COMM_WORLD);

（以下右上に続く）
```

```
     for(i=k+1;i<n;i++)  {
        if( MAP[i] == my_rank )
          for(j=k+1;j<n;j++)
           a[i][j] -= a[i][k] * a[k][j];
     } /* end of i-loop */
  } /* end of k-loop */

 /* copy a[i][*] of my_rank ->  a[i][*] of Rank0  */

     for(i=0;i<n;i++)  {
       if(MAP[i] > 0) MPI_Send(&a[i][0],n,
      MPI_DOUBLE,0,i,MPI_COMM_WORLD);
     }
     if(my_rank==0)
       for(i=0;i<n;i++) {
       if(MAP[i] > 0 ) MPI_Recv(&a[i][0],n,
              MPI_DOUBLE,MAP[i],i,
              MPI_COMM_WORLD,&status);
       }
} /* End of lu_p */
```



並列化方法の違いによる実効性能の差

- 並列化方式の違いによる性能差の要因
  - 各CPUへ割り振るタスクの平均化
  - 1 CPU内でのキャッシュの利用効率
  - 通信量の影響(今の場合はほとんど無関係)
- block分割のメリット・デメリット
  - タスクが十分に平均化されない可能性。
  - キャッシュの利用効率は比較的よい
- cyclic分割のメリット・デメリット
  - タスクの平均化はかなりよい。
  - キャッシュの利用効率は悪い。
- block-cyclic分割
  - blockとcyclicのいいとこ取り??
  - タスクの平均化が十分であれば、連続した領域のほうがよい。



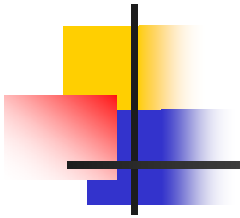
実行時間の計測について

◆ 所要時間（Turn Around Time）

最初に実行を開始したプロセスの開始時刻から、最後に実行を終了したプロセスの終了時刻までを計測し，所要時間とする.

```
MPI_Barrier(MPI_COMM_WORLD);
  t_start = MPI_Wtime();
    /*この間に所要時間を測定する処理を記述*/
  MPI_Barrier(MPI_COMM_WORLD);
  t_stop = MPI_Wtime();

printf("Turn around time = %.16f¥n",
  t_stop  -  t_start  );
```

MPI_Barrier
「バリア同期」と呼ばれ、すべてのプロセスがこれを呼び出すまで各プロセスが待ち合わせる機能を持つ，MPI標準の関数.



PCクラスタを利用した演習（12月20日）

利用計算機：

Xeon 3.06GHz dual processor 16 台から成るPCクラスター

Memory:4GB/node, Disk: 1.2TB(total)

Login UID:

講義中に配布

端末からのLogin 方法：

Windows XP telnet or sshで omega.cc.kyushu-u.ac.jp へ接続

File 転送：Local ⇔ PC Cluster

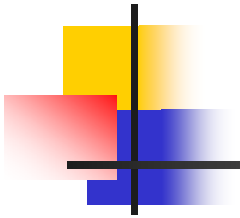
IE(ブラウザ）からftp://UID:PWD@omega.cc.kyushu-u.ac.jp/

File 編集：LocalのWindows上

メモ帳，ワードパッド等，エディター

File 編集：PC Cluster上

vi エディター



PCクラスタを利用した演習(cont.)

Login後の設定:

PCクラスタにはScoreという並列環境が入っており，これをactivateするためにコマンドプロンプトから scout –g all と入力すること.

Mpi program のコンパイル＆リンク:

mpicc　test.c

Mpi program の実行:

mpirun –np [1-16のノード数] a.out　[programの引数]

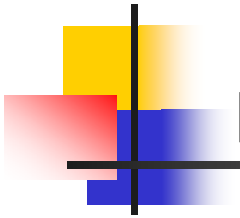
はじめの一歩:

講義で指定したLU分解の並列計算プログラムを翻訳＆実行する

実行時間の計測:

MPI_Barrier(COMM), MPI_Wtime() 関数を使って，時間を計測できるようにprogramを変更し，np 数を変えて経過時間を計る.

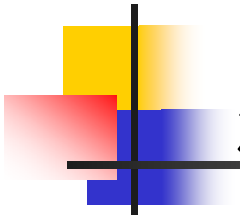
MPI_Wtime() 関数の型は　double .



PCクラスタを利用した演習(cont.2)

今日の演習内容

- 問題サイズと所要時間
  - どの程度の時間になるか予測する。
  - 問題サイズを変えて経過時間を測定する。
- プログラムの並列化
  - 関数 lu_f() のMAP[i]という配列の指定方法を変更し、block/cyclic/block-cyclic分割を試す。
  - 結果が変わらないことを確認しておく。
- 並列化による効果を確認する
  - 横軸に並列CPU数をとり、所要時間の表を作成。



演習レポート(12月13日 出題) 再掲

【課題】

講義で使用した行でcyclic分割したLU分解プログラムを基に,
以下について並列化効率の観点から考察しレポートにして,
下記の要領でメールで提出してください.

(1) LU分解部分の経過時間を計測できるように書き換え, 使用ノード数を変えて計測結果を表またはグラフにまとめる.

(2) 行でBlock分割するようにLU分解プログラムを変更し, 上と同様に経過時間を計測し結果を(1)と比較する.

(3) 行でBlock-Cyclic分割するようにLU分解プログラムを変更し, (1)(2)と同様に経過時間を計測し結果を(1)(2)と比較する.

提出先:aoyagi@cc.kyushu-u.ac.jp
Subject: 並列アルゴリズム課題
締め切り:平成17年12月27日(火)

形式:text, pdf, ps, Word, Excel ファイルの添付可