［大規模科学計算システム］

並列コンピュータ Express5800/A1080a-D の並列処理

吉田　正浩
日本電気株式会社　HPC 事業部

１．はじめに

HPC の分野では、複数の CPU を使って並列実行し高速化を図ってきました。その中でも、SX-9 のように超高速な CPU（もしくはコア）により、逐次実行での性能を確保したうえで、さらに並列計算も行いより一層の性能向上を図るというアプローチと、多くのコアで並列計算を行うことで処理速度を向上させるというアプローチがあります。サイバーサイエンスセンターで運用している並列コンピュータ Express5800/A1080a-D は、１つの計算ノードに 8 個のコアを搭載した CPU を 4 台搭載することで 32 コアが利用可能となっており、後者のアプローチに適した計算機だと言えます。

Express5800/A1080a-D は共有メモリ型の計算機であり、そのアーキテクチャは図 1 のように表されます。メモリはメモリノードという各 CPU に直結されたブロックに分かれており、共有メモリ型計算機の中でも特に NUMA(Non-Uniform Memory Access)と呼ばれるアーキテクチャとなっています[1]。

この資料では並列計算の入門として、Express5800/A1080a-D で利用可能な並列計算の手法のうち、自動並列化と OpenMP[2]による並列化をご紹介します。

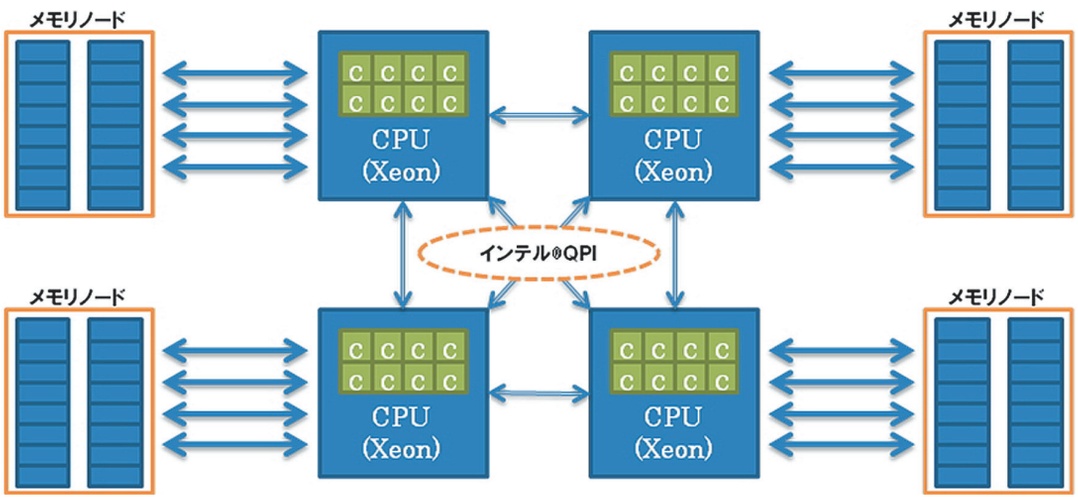


図 1 Express5800/A1080a-D のアーキテクチャ

２．並列処理の概要

一つのジョブを複数の CPU やコアで分散して処理するために、様々な手法・規格が定められています。これらを分類すると、共有メモリ型の計算機での並列実行を念頭においた「スレッドレベルでの並列化」と分散メモリ型の計算機で並列実行を行うための「プロセ

スレベルの並列化」の二種類に分けられます。

スレッドレベルの並列化では、並列に実行が可能な do ループ等の処理を、並列に実行可能な単位である「スレッド」に分割し、複数のスレッドを各コアに割り当てる事で並列実行を実現しています。プロセスレベルの並列化では、複数の「プロセス」を起動し、各プロセスが割り当てられた処理を並列に実行します。並列実行の制御とデータの転送のために、プロセス間の通信を行う必要があります。

図 2 に逐次実行（非並列実行）、スレッドレベルの並列実行、プロセスレベルの並列実行の概念図を示します。スレッドレベル、プロセスレベルのどちらの並列実行においても、複数のコアを使う点は同じですが、スレッドレベルが単一計算ノード内の複数のコアを使用する点に限られるのに対し、プロセスレベル並列化ではノード間にまたがった並列化が可能です。

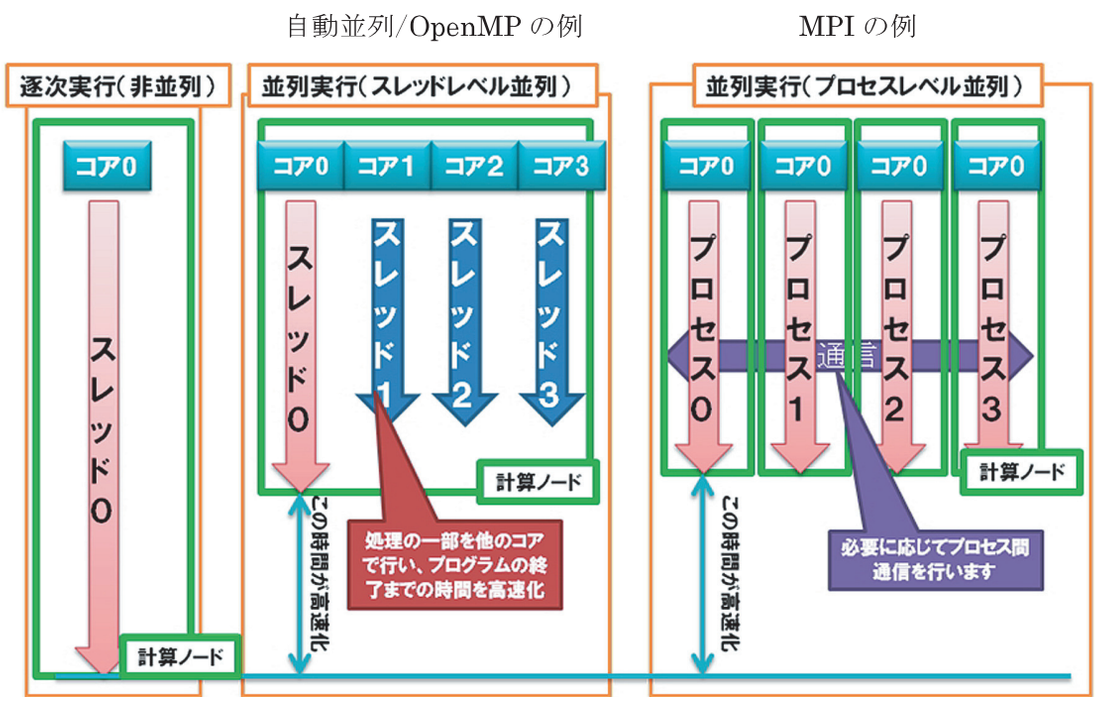


図 2 並列実行モデル

現在、広く利用されている並列化の手法としては、自動並列、OpenMP、MPI などがあげられ、自動並列化と OpenMP はスレッドレベルの並列化を意図した手法であり、MPI のみがプロセスレベルの並列化が可能な手法です。

自動並列化は、コンパイラが並列化可能な処理を認識し、自動的にスレッドに分割可能とする手法です。コンパイラオプションを指定するだけで、複数のコアを使った並列計算が可能となります。ただし、プログラムの構造が複雑な場合には、期待通りに並列化が行われず、性能向上がみられない場合もあります。一方、OpenMP はプログラマが並列実行する部分をコンパイラ指示行で指定することで、スレッドレベルの並列化を行います。

OpenMP は指示行による並列化を実現する API（Application Program Interface）で、プログラム中に指定する指示行の意味や、並列化をサポートする専用の関数などが定義されています。

以降では Fortran のプログラムリストを基に解説を行います。一部 C 言語のプログラムリストも併記しています。

３．自動並列化

自動並列化は、コンパイラが“安全”に並列化可能だと判断したループや処理のうち、高速化が期待できる部分を自動的にスレッドに分割する並列化方法です。“安全”というのは、並列に実行しても浮動小数点演算による丸め誤差を除いて、計算結果に影響を与えないという意味となります。リスト 1 のループは、異なる i の値に対しての処理が独立しているために、i の値毎にスレッドに分割しても安全だと判断できます。一方、リスト 2 では、i の処理に i-1 の値を使っており、異なるループの処理内容が他のループの処理結果に影響を与える「依存関係」が存在しています。この場合、コンパイラは安全では無いと判断し、自動並列化による並列化を行いません。

```
do i=1,1000
  a(i) = b(i) * c(i)
enddo
```

リスト 1

```
do i=2,1000
  a(i) = b(i) * a(i-1)
enddo
```

リスト 2

センターで提供している、Intel Fortran,C/C++コンパイラでは、“-parallel”をコンパイラオプションに指定することで自動並列化が利用可能です。コンパイル方法は図 3 の通りです。この際に、“-par-report”オプションも同時に指定することで、プログラム中のどのループが並列化されるかを表示できます。さらに詳細情報を出力させる“-par-report=3”を指定すると、並列化されていないループが、何故並列化されなかったかも出力されます。図 3 のコンパイル例では、プログラム中 12 行目のループは並列化されているが、15 行目のループは依存関係があるために並列化されていないというメッセージが出力されています。

並列処理を実行するには、通常プログラムの実行時に環境変数 OMP_NUM_THREADS で並列数を指定しますが、センターで提供しているジョブ管理機能では、ジョブクラスによって自動的に並列数が設定され、ジョブクラス a8, a16, a32 ではそれぞれ 8, 16, 32 の並列数で実行可能です。図 4 の並列数の指定例では 4 並列で実行されます。詳細は並列コンピュータ Express5800 の利用法[4]を参照ください。

ループの自動並列化を阻害する主な要因として次のような場合があります。このような場合には、プログラムを書き換えていただくか、指示行を挿入することでコンパイラに並列化可能であることを示すことで、自動並列化が可能となる場合もあります。

・ループ中に依存関係が存在する場合
・ループの繰り返し回数が少なく、自動並列化の性能向上が見込まれない場合
・ループの繰り返し回数がループ開始時に予め分からない場合（while ループなど）
・ループ内部へのジャンプや、ループ外部へのジャンプがある場合（goto 文など）
・サブルーチンや関数の呼び出しがある場合（write 文、read 文などの IO も含む）
・依存関係を判断できない間接参照（またはポインタの参照）が存在する場合

```
[Fortran]
f95 –parallel –par–report=3 program.f90
program.f90(12): (col. 3) remark: LOOP WAS AUTO-PARALLELIZED
program.f90(15): (col. 3) remark: loop was not parallelized:
                                  existence of parallel dependence.

[C 言語/C++]　コンパイル方法
cc –parallel –par–report=3 program.c
```

コンパイル時のメッセージ

図 3 コンパイル方法とコンパイルの様子

```
[csh の場合の例]
setenv OMP_NUM_THREADS 4
./a.out
[bash の場合の例]
export OMP_NUM_THREADS=4
./a.out
```

図 4 並列数の指定と実行

リスト 3 に示すプログラムで、自動並列化の動作を検証します。このプログラムを自動並列化オプション付きでコンパイルし、4 つのスレッドで実行すると、図 5 のようなイメージで処理が分割され実行します。スレッドレベル並列化の際に、それぞれのスレッドは固有のスレッド ID を持ちます。自動並列化の場合は特に意識する必要はありませんが、図 5 では、以降の説明のためにスレッド ID の例も記載しています。

並列処理の様子を具体的に考察します。初めに、スレッド 0（マスタースレッドとも呼ばれます）が単独でプログラムの実行を開始します。自動並列化によって並列化されたループに到達すると、スレッド 0 によって、3 つのスレッドが生成されます。スレッド 0 を含めた各スレッドには、ループのインデックス i の担当範囲が定められており、それぞれが割り

当てられたiの担当範囲の処理を開始します。今、スレッド0がa(1) = b(1) * c(1)という処理を始めるとすると、ほぼ同時にスレッド3がa(751) = b(751) * c(751)の処理を行っていると考えられます。スレッド0はa(1)に関する処理を完了するとa(2)=b(2)*c(2)を、スレッド3はa(751)の処理が完了すると次にa(752)=b(752)*c(752)を…という順序でそれぞれのスレッドが独立して、処理を進めます。各スレッドは、自身に割り当てられた処理が完了すると、他のスレッドの処理完了を待ち、全てのスレッドがそろった時点で同期をとり並列処理を終了します。

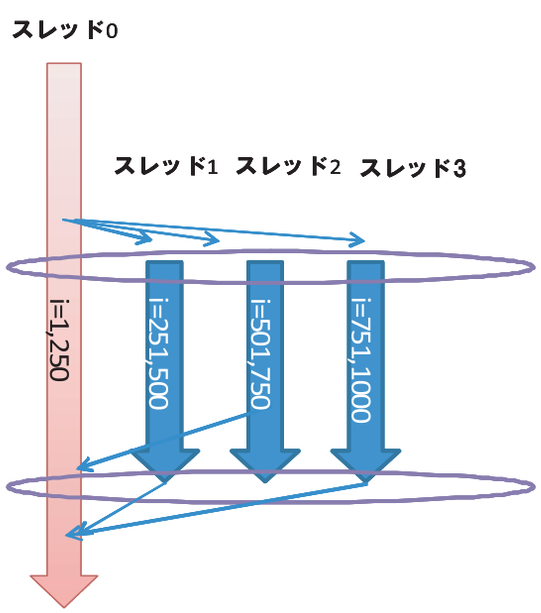
```
subroutine auto_parallel(a, b, c)
   integer :: i
   real(8),dimension(1000) :: a, b, c
   do i=1,1000
      b(i) = i
      c(i) = 2*i
   enddo
!乗算
   do i=1,1000
     a(i) = b(i) * c(i)
   enddo
end subroutine auto_parallel
```

Fortran

```
void loop_parallel(double *a, double
*b, double *c){
   int i;
   for(i=0;i<1000;i++){
      b[i] = i;
      c[i]= 2*i;
   }
//乗算
   for(i=0;i<1000;i++){
     a[i] = b[i] * c[i];
   }
return;
```

C

リスト 3



スレッド0(マスタースレッド)のみで処理を開始します

スレッド0がスレッド1,2,3を作成します

各スレッドが異なるiの範囲の処理を行います

並列処理が終わると、スレッド1,2,3を破棄して、スレッド0のみで残りの処理を続けます

図 5 自動並列化による並列実行イメージ

4．OpenMP

OpenMP は、プログラマが指示行を挿入することで並列化を行う手法です。プログラマの責任で並列処理を指示するため、自動並列化ではコンパイラが“安全”に並列化できると判定できないループや処理についても並列化が可能です。逆に、**アルゴリズム上並列化できないような処理に関しても、誤って OpenMP の指示行を追加すると強制的に並列化されてしまい、結果不正を引き起こすことがあるので注意が必要です。**

4.1 OpenMP による並列化の例

初めに、リスト 4 に示す簡単な OpenMP のプログラムの実行を通して、OpenMP 使用の流れを解説します。このプログラムは、並列実行時に自らのスレッド ID を出力するという動作を行います。

```
program write_myid
implicit none
integer::myid
integer::omp_get_thread_num
myid=0

!$omp parallel private(myid)
   myid = omp_get_thread_num()
   write(*,*) "My thread id is ",myid
!$omp end parallel
end program write_myid
```

Fortran

```
#include <stdio.h>
#include<omp.h>
int main(){

int myid=0;
#pragma omp parallel private(myid)
{
   myid = omp_get_thread_num();
   printf("My thread id is %d¥n", myid);
}
}
```

C

リスト 4

OpenMP の指示行は、Fortran の固定形式では「Comp」、Fortran の自由形式では「!$omp」、C 言語では「#pragma omp」から始まる行で指定されます。このプログラムは、指示行により!$omp parallel から !$omp end parallel で囲まれた部分を並列実行可能と指定しています。この部分を並列領域と呼び並列領域に含まれる処理が、実行時に指定されたスレッド数で並列に実行されます。このプログラムは並列領域で、omp_get_thread_num 関数を呼び出しています。この関数は OpenMP の規格で定義された補助関数であり、スレッドにユニークに割り当てられた番号、すなわち、スレッド ID を返します。各スレッドが取得したスレッド ID を次の行で出力し、並列領域を終了しています。

このプログラムは図 6 のようにコンパイラオプションに“-openmp”を指定してコンパイルします。OpenMP によって、並列化されたプログラムも、自動並列化と同様に OMP_NUM_THREADS 環境変数で並列数を指定します。実行は図 7 の通り、通常のプロ

グラムと同様に./a.out という形で行います。このプログラムの実行イメージは、図 8 のようになっています。

```
[Fortran]
f95 –openmp    program .f90
[C 言語/C++]
cc    –openmp program.c
```

図 6 コンパイル方法(OpenMP)

```
$ f95 -openmp write_myid.f90
$ setenv OMP_NUM_THREADS 4
$ ./a.out
 My thread id is        0
 My thread id is        3
 My thread id is        1
 My thread id is        2
```

Fortran

```
$ cc -openmp write_myid.c
$ setenv OMP_NUM_THREADS 4
$ ./a.out
 My thread id is        0
 My thread id is        2
 My thread id is        1
 My thread id is        3
```

C

図 7 実行方法と実行結果

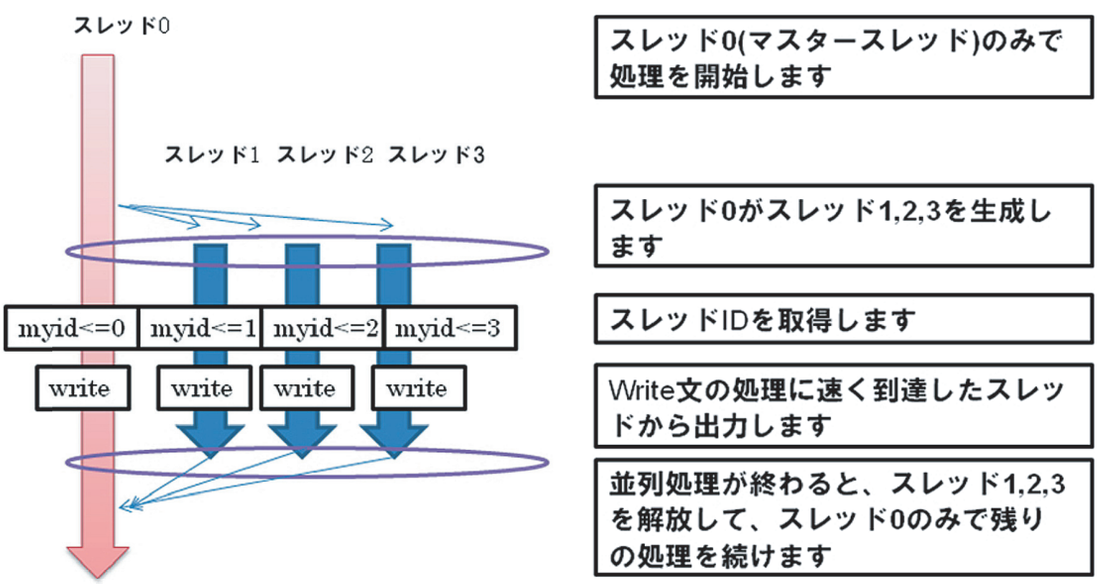


図 8 プログラムの実行イメージ

結果を確認します。図 7 において、4 行の出力が行われていることより、4 並列で動作している事が確認できます。また、ID が 0,1,2,3 の順ではなく、0,3,1,2 と出力されており、各スレッドの処理される順序がスレッド ID 順ではない事が確認できます。これは、（何らかの同期を行わない限り）スレッドの実行順序は保証されず、先に write に辿りついたスレッドから ID を出力するためであり、このプログラムでは、実行する度に出力される順序が変わります。OpenMP で並列化した場合は、各スレッドに割り当てられた処理を独立して実行するため、実際に処理が行われる順序についての保証がなく、結果が非並列実行と異なる場合があります。OpenMP で並列化を行う際はこの点に注意が必要です。（スレッド内の処理は順に実行完了します。）

4.2 ループの並列化

ここでは、ループの並列化方法を述べます。ループの並列処理を行う場合は!$omp parallel do 指示行を用います。この指示行を挿入すると、直後の do ループが OMP_NUM_THREADS 等で指定されたスレッド数に応じて分割されます。

```
subroutine loop_parallel(a, b, c)
  integer :: i
  real,dimension(1000) :: a, b, c
!$omp parallel do
  do i=1,1000
    a(i) = b(i) * c(i)
  enddo

end subroutine loop_parallel
```

Fortran

```
void loop_parallel(double *a, double
*b, double *c, int N){
  int i;
#pragma omp parallel for
for(i=0;i<N;i++){
      a[i] = b[i] * c[i];
  }
return;
}
```

C

リスト 5

リスト 5 のプログラムをコンパイルし実行すると、図 5 で示した自動並列化の例と同様に動作します。!$omp parallel do 指示行は、スレッドの生成、解放とループの分割まで行うため、前節の!$omp parallel と!$omp end parallel の指定は不要です。また、parallel do 指示行は直後の do ループに対して有効であり、リスト 6 の(a)と(b)では、分割されるループが異なります。(a)では j のループが、(b)では i のループが分割されます。(b)のループでは、ループの分割処理を j のループ回数行う必要があり、一般に(a)よりもオーバーヘッドが大きくなります。

```
!$omp parallel do
 do j=1,1000
  do i=1,1000
     a(i,j) = b(j) * c(i)
  enddo
enddo
```

(a)

```
 do j=1,1000
!$omp parallel do
   do i=1,1000
      a(i,j) = b(j) * c(i)
   enddo
enddo
```

(b)

リスト 6

4.3 private 変数と shared 変数

OpenMP では、変数の属性を指定できます。属性の指定は!$omp parallel や!$omp parallel do に続けて、“属性名(変数名)”という形で指定します。リスト 4 の例では!$omp parallel do private(myid)と myid を private 変数に指定していました。

よく使用される属性は private と shared の二つの属性です。private 属性を持つ変数（これを private 変数と呼びます）は、スレッドごとに独立した値を保持できる変数で、一時的な変数などに用いられます。private 変数の初期値、終了値は不定となります。よって、並列処理部内で値の代入を行う必要があります。

また、shared 属性を持つ変数(shared 変数と呼びます)は、すべてのスレッドからアクセス可能な変数です。すべてのスレッドが参照・更新できてしまうため、更新は他のスレッドと排他的に行う必要があります。

OpenMP の並列領域内の変数はデフォルトで shared 変数になります。ただし、!$omp do で並列化したループの制御変数と、Fortran の do ループの制御変数は自動的に private 変数となります。注意が必要なのは、C 言語で 2 重ループの外側ループで並列化する場合で、内側ループの制御変数を明示的に private 変数と指定しなければ、制御変数が shared 変数とみなされ、期待した実行結果が得られなくなります。

リスト 7 に、tmp という変数を private 変数として使用する例を示します。このプログラムでは、変数 tmp はスレッドごとに保持されます。スレッド 0 の tmp とスレッド 1 の tmp は全く別の変数として扱われ、独立した値を保持します。他のスレッドのプライベート変数に直接アクセスすることはできません。もしも、tmp が shared 変数として扱われた場合、複数のスレッドが同時に更新しようとするため、更新後の値が最後に書き込んだスレッドの値となり、次の参照時に意図しない値を読み込むことになります。

```
!$omp parallel do private(tmp)
do i=1,n
   tmp = 0.0d0
   do j=1,m
      tmp = tmp + a(j,i)
   enddo
   b(i) = tmp
enddo
```

Fortran

```
#pragma omp parallel for private(tmp,j)
for(i=0;i<n;i++){
   tmp = 0.0;
   for(j=0;j<n;j++){
      tmp = tmp + a[i][j];
   }
   b[i] = tmp;
}
```

C

リスト 7

5. ファーストタッチ

OpenMP の紹介の最後に、NUMA アーキテクチャの計算機向けに考慮が必要となるファーストタッチと呼ばれる手法をご紹介します。先に述べたとおり、Express5800/A1080a-D は NUMA アーキテクチャの共有メモリ計算機です。NUMA アーキテクチャでは、ある CPU から見て直結されたメモリノードをローカルメモリ、他の CPU 配下のメモリノードをリモートメモリと呼びます。リモートメモリへのアクセスを行うと、ローカルメモリへのアクセスに比べて、メモリレイテンシ（メモリアクセス時の遅延時間）が伸び、さらに、CPU 間を接続している QPI(QuickPath Interconnect)がボトルネックになる恐れがあります。また、図 9 のように、特定のメモリノード間にアクセスが集中すると、アクセスが集中したメモリノードと CPU 間のデータ転送がボトルネックとなり、並列性能が低下する原因となります。

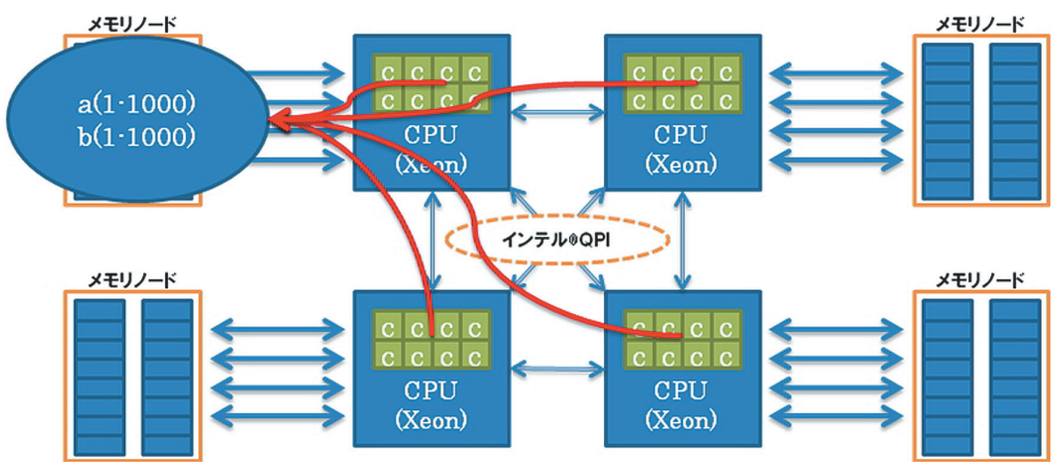


図 9 メモリアクセスの集中

そこで、プログラム中で使用する配列がどのメモリノードに配置されるかを制御する必

要が生じます。Express5800/A1080a-D をはじめ、ほとんどの NUMA マシンでは、変数・配列を初期化した（初めに値を書き込んだ）コアのローカルメモリに優先して配置するよう制御されます。この仕様を利用して、配列の初期化部分を並列化する事で、図 10 のように配列の配置を各メモリノードに分散可能です。この手法により、リモートメモリへのアクセスを防ぐと同時に、一つのメモリノードへのアクセス集中を回避することができます。配列の初期化を並列に行い、メモリアクセスをローカルメモリ中心となるように制御する手法をファーストタッチと呼びます。

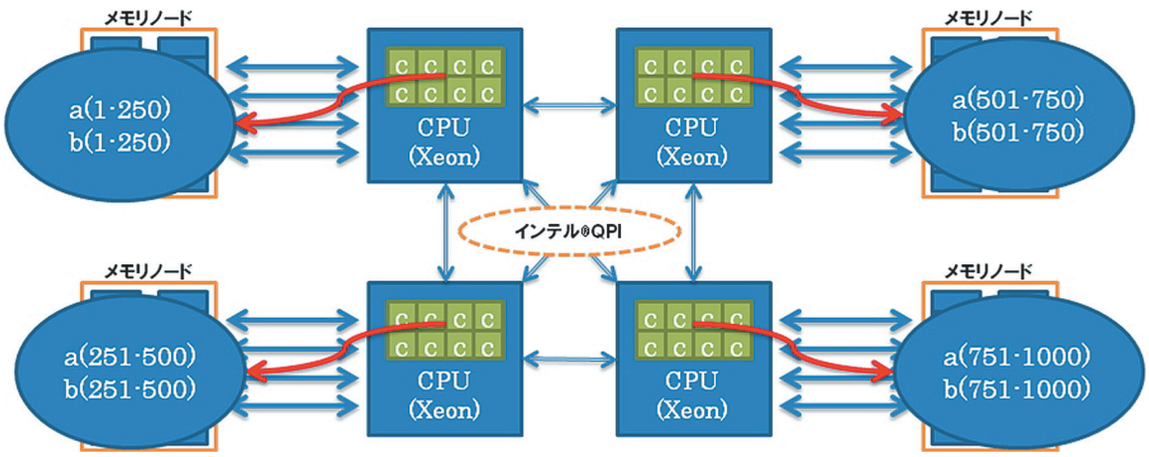


図 10 ファーストタッチ後のメモリアクセス

ファーストタッチの例として、次のリスト 8 のプログラムで考察します(なお、このプログラムは stream ベンチマーク[3]を模しています)。このプログラムでは、サブルーチン Triad の j のループで並列化でき、処理時間のほとんどは Triad によって占められます。それゆえ、メモリの配置を考えなければ、Triad の他の部分を並列化する必要はありません。しかし、ファーストタッチを行わず、Triad のみを並列化した場合では、表 1 のように 32 並列で 2.9 倍の性能向上にとどまっており、効率的な並列化が出来ているとは言えません。

表 1 ファーストタッチの効果

	ファーストタッチ無し		ファーストタッチ有り
並列数	1	32	32
サブルーチン部分(秒)	114.2	39.3	6.8
プログラム全体(秒)	114.8	40.0	7.1
サブルーチン部分性能向上率 (1 並列を 1.0 とした場合)	1.0	2.9	16.8

```
program stream
implicit none
integer:: N,NTIMES,j,k
parameter (N=20000000,NTIMES=1000)
real(8)::a(N),b(N),c(N),scalar
real(8)::omp_get_wtime,time,time1,time2
   do j=1,N
    a(j)=1.0
    b(j)=2.0
    c(j)=3.0
   enddo
   scalar=1.0d0
   time=0.0d0
   do k=1,NTIMES
    time1 = omp_get_wtime()
    call Triad(scalar,a,b,c,N)
    time2 = omp_get_wtime()
    time = time + (time2 - time1)
   enddo
   do j=1,N
    scalar = scalar + a(j)
   enddo
   write(*,*) scalar,time
end program stream

subroutine Triad(scalar,a,b,c,N)
integer:: N,j
real(8)::a(N),b(N),c(N),scalar
!$omp parallel do
   do j=1,N
     a(j) = b(j)    + scalar*c(j)
   enddo
end subroutine Triad
```

Fortran

```
#include <stdio.h>
#include <omp.h>
#define N 20000000
#define NTIMES 1000
void Triad(double scalar);
static double a[N],b[N],c[N],scalar;
int main(){
      int k,j;
      double time,time1,time2;
      for (j=0; j<N; j++) {
            a[j] = 1.0;
            b[j] = 2.0;
            c[j] = 3.0;
            }
      scalar=1.0;
      time=0.0;
      for (k=0; k<NTIMES; k++) {
            time1=omp_get_wtime();
            Triad(scalar);
            time2=omp_get_wtime();
            time=time+(time2-time1);
            }
    for (j=0; j<N; j++) {
                scalar=scalar + a[j];
            }
    printf("%f %f¥n",scalar,time);
}
void Triad(double scalar){
            int j;
#pragma omp parallel for
            for (j=0; j<N; j++)
                  a[j] = b[j]+scalar*c[j];
}
```

C

リスト 8

そこで、リスト 9 では初期化部分も並列化し、ファーストタッチを行います。配列 a,b,c ともに、初めてのアクセスは初期化部で行われ、各スレッドが Triad 内で処理を担当するインデックス”j”の範囲の値を初期化しています。これにより a,b,c の各要素は図 10 のように、初期化を担当したスレッドが割りつけられたコアのローカルメモリに配置され、アクセスの集中とリモートメモリへの頻繁なアクセスを回避できます。ファーストタッチの性能は表 1 に示した通りであり、32 並列での性能向上が確認できています。

```
(宣言部省略)
!$omp parallel do
   do j=1,N
      a(j)=1.0
      b(j)=2.0
      c(j)=3.0
   enddo
(中略)
subroutine Triad(scalar,a,b,c,N)
(中略)
!$omp parallel do
   do j=1,N
      a(j) = b(j)    + scalar*c(j)
   enddo
end subroutine Triad
```

Fortran

```
(宣言部省略)
#pragma omp parallel for
      for (j=0; j<N; j++) {
            a[j] = 1.0;
            b[j] = 2.0;
            c[j] = 3.0;
 (中略)
void Triad(double scalar){
 (中略)
#pragma omp parallel for
            for (j=0; j<N; j++)
                  a[j] = b[j]+scalar*c[j];
}
```

C

リスト 9

ファーストタッチを行う際に、演算部分の並列ループと初期化時の並列ループの“メモリアクセスパターン”を可能な限り近づける必要があります。リスト 10 では、配列 a(i,j) の初期化を j のループで行っていますが、演算部分では内側の i のループで並列化し a(i,j) にアクセスします。このとき、a(i,j)は j で分割されメモリに配置されているため、演算部では図 11 のようにリモートメモリへのアクセスが頻繁に行われます。このようなケースでは、初期化をリスト 11 のように演算ループと同じく i のループで並列化することで、演算部分でのリモートメモリへのアクセスを大幅に削減が可能となります。

```
program memorytest
implicit none
integer:: N,i,j,k
parameter (N=2000)
real(8)::a(N,N),b(N),c(N,N),scalar
!$omp parallel do
   do j=1,N
    do i=1,N
     a(i,j)=2.0d0
     c(i,j)=1.0d0
    enddo
   enddo
!$omp parallel do
   do j=1,N
     b(j)=1.0d0
   enddo
   call mem_test(a,b,c,N)
   scalar = 0.0d0
   do j=1,N
       scalar = scalar + c(N,j)
   enddo
   write(*,*) scalar
end program memorytest

subroutine mem_test (a,b,c,N)
integer:: N,i,j
real(8)::a(N,N),b(N),c(N,N)
   do j=2,N
     b(j) = c(1,j-1)
!$omp parallel do
      do i=1,N
        c(i,j) = c(i,j) + a(i,j) / b(i)
      enddo
   enddo
end subroutine mem_test
```

リスト 10

```
program memorytest
implicit none
integer:: N,i,j,k
parameter (N=2000)
real(8)::a(N,N),b(N),c(N,N),scalar
   do j=1,N
!$omp parallel do
     do i=1,N
      a(i,j)=2.0d0
      c(i,j)=1.0d0
     enddo
   enddo
!$omp parallel do
   do j=1,N
     b(j)=1.0d0
   enddo
   call mem_test(a,b,c,N)
   scalar = 0.0d0
   do j=1,N
       scalar = scalar + c(N,j)
   enddo
   write(*,*) scalar
end program memorytest

subroutine mem_test (a,b,c,N)
integer:: N,i,j
real(8)::a(N,N),b(N),c(N,N)
   do j=2,N
     b(j) = c(1,j-1)
!$omp parallel do
      do i=1,N
        c(i,j) = c(i,j) + a(i,j) / b(i)
      enddo
   enddo
end subroutine mem_test
```

リスト 11

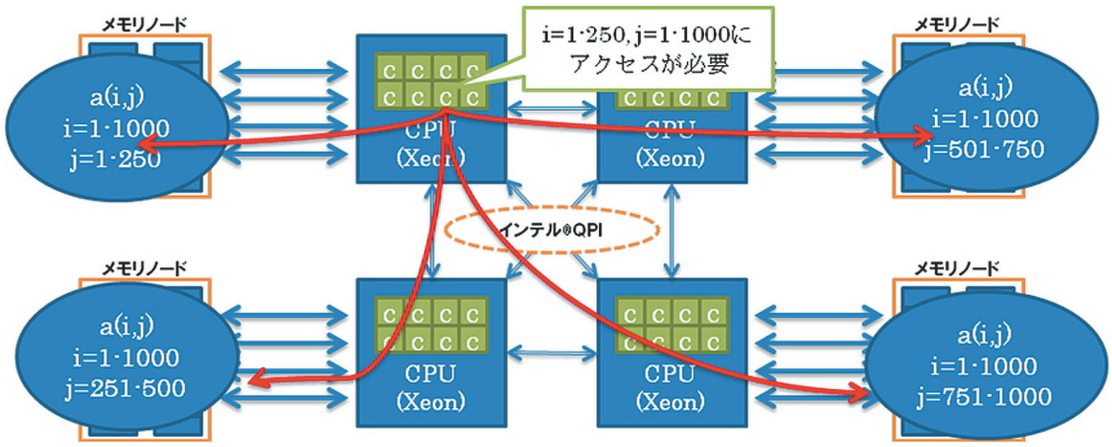


図 11 初期化と異なるループで並列化を行った場合のメモリアクセスパターン

6. おわりに

並列計算の入門として、自動並列化や OpenMP をご紹介いたしました。自動並列化、OpenMP ともに、紹介しきれなかった機能や、技法が存在します。より高度な機能については Intel コンパイラのマニュアルを参照ください。

並列計算はハードルが高いと敬遠される方もいらっしゃいますが、うまく利用できると数十倍の高速化を達成できる場合もあります。本稿が、並列化を行うきっかけや、並列計算を行う手助けになれば幸いです。

参考文献

[1] 那須 康之, 鈴木 健一, 谷岡 隆浩 “Express5800/A1080a-D のハードウェア”, SENAC Vol.43, No. 3(2010.7)

[2]OpenMP Architecture Review Board “OpenMP Application Program Interface Version 3.0 May 2008”, http://www.openmp.org/mp-documents/spec30.pdf

[3] John D. McCalpin, “STREAM:Sustainable Memory Bandwidth in High Performance Computers”, http://www.cs.virginia.edu/stream/

[4] 情報部情報基盤課共同研究支援係, 共同利用支援係, サイバーサイエンスセンター スーパーコンピューティング研究部 “並列コンピュータ Express5800 の利用法”, SENAC Vol.43, No. 2(2010.4) ,http://www.ss.isc.tohoku.ac.jp/service/gen/Express-Ver3.pdf