SSTC
High Performance and Productivity
スケーラブルシステムズ株式会社

Cluster OpenMP のご紹介

~ プログラム開発のワークフローについて

スケーラブルシステムズ株式会社

Cluster OpenMP のご紹介
~ プログラム開発のワークフローについて

スケーラブルシステム株式会社では、IT 技術と HPC システムに関する様々な調査レポートを発行しています。

2007.01.05

はじめに

多くの分野において、科学者や技術者は、重要で複雑な問題を解くためにハイパフォーマンスコンピューティングシステム（HPC システム）を利用しています。マイクロプロセッサが、その動作クロックを向上させ、64 ビットアドレッシングをサポートすることで、デスクトップもその処理能力を大幅に向上させています。しかし、並列処理を基本とする HPC システムの利用も拡大しています。その大きな理由として、以下のような要因があります。

1） 解析対象のモデルやアルゴリズム、解析現象の複雑化のため、デスクトップでは物理的に解析できない問題が増加している
2） シングルプロセッサのクロックアップでの性能向上の限界に対処する為にプロセッサのマルチコア化とそのようなプロセッサを複数搭載したマルチプロセッサシステムの利用が必須となっている
3） マルチプロセッサを搭載する並列計算機システムがより廉価に導入できることようになっている

このように HPC システムの利用が一般化することで、従来よりもより深刻化している問題があります。それが、「ソフトウエアのギャップ」と「プログラム会社のワークフロー」の問題です。

ソフトウエアのギャップ

並列計算機を活用する並列プログラミングは、マルチプロセッサを搭載した並列計算機システムが一般化しているにも関わらず、依然として容易ではありません。このことは、並列計算機システムが一般化し、また、システムがより高速化されていることにより、プログラミングと並列計算機システムの能力のギャップの拡大を引き起こすことになっています。
このようなギャップを埋めることは、今後の並列計算機システムの発展を考えても急務となっています。また、並列計算機システムの複雑な構成を意識することなく、プログラミングを可能とすることは、プログラム開発の生産性の向上にも大きく寄与します。

プログラム開発のワークフロー

図-1に示すようなワークフローが典型的な HPC システムでのアプリケーション開発のひとつです。プログラムのアルゴリズムやモデリングをテストする場合、デスクトップなどのより身近なシステムでの開発を行い、その後で、HPC システムでのテストや実データでの検証などを行なうことになります。このようなワークフローには、柔軟性や効率化という点では、常に問題があります。デスクトップでのプロトタイプで利用する計算機環境やプログラミングモデルと HPC システムでのプログラミングモデルは必ずしも一致しないケースもあり、また、HPC システムでの高

いスケーラビリティを実現するには、高度な知識とプログラミング技術が求められる場合があります。そのような技術はある意味、特殊な技術を持つスペシャリストを必要としますが、そのようなスペシャリストは非常に限られています。また、マルチプロセッサでのプログラム開発では、プログラムのデバッグや実行時のボトルネックの解析は、HPC システムの構成規模が大きく、より多くのリソースを利用するような場合には、非常に困難なタスクとなります。また、小規模な問題やテスト的に利用するモデルでは問題なく解析出来ても、本来のシュミレーションで要求されるデータやモデルでは問題が発生する場合もあります。

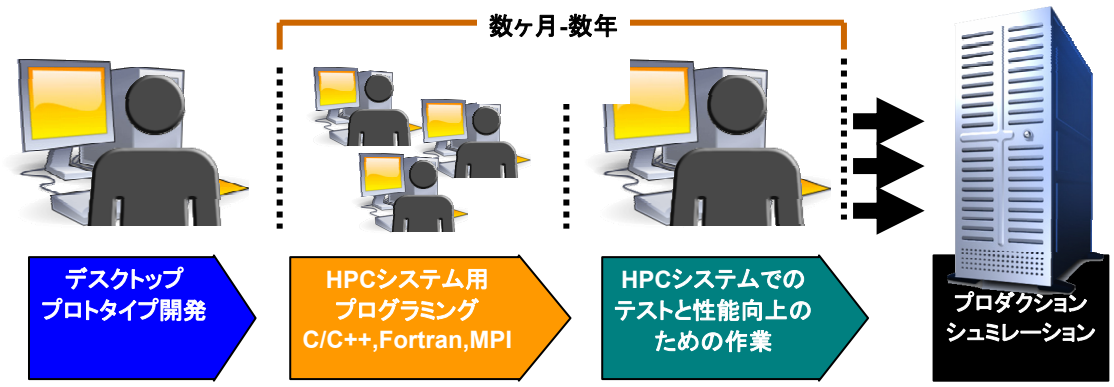

図-1　従来型の並列アプリケーションの開発プロセス　（バッチワークフローでの開発プロセス）

プログラムの開発段階では、プログラム自身のコーディングはもちろん、アルゴリズムやアプリケーションの仕様なども頻繁に変更になります。このような開発には、長い開発期間を必要とすることになり、コストも非常にかかることになります。このようなプログラムの開発が終了した後でのシュミレーションは、一般には‘ジョブ’として、HPC システムが用意しているバッチシステムで実行されることになります。バッチシステムでの実行は、プログラムの開発やテストで行ってきたような対話処理ではないため、その実行結果の確認はジョブの終了を待つことになります。

エンジニアや科学者が必要とするのは、結局は、‘Time-to-Solution （問題解決までの時間）’の短縮であることから、このようなプログラムやソリューションの開発期間も含んだトータルなワークフローでの時間短縮や効率化が求められています。

この資料では、このような「ソフトウエアのギャップ」と「プログラム開発のワークフロー」の問題に対する一つのソリューションとして、OpenMP による共有メモリ並列プログラミングとこの OpenMP をクラスタ環境で利用可能とする Cluster OpenMP について示します。

OpenMP による共有メモリ並列プログラミング

OpenMP は、共有メモリ並列計算機システムでの並列プログラミングのために活用されています。OpenMP は最初、対称型並列計算機 Symmetric Multi-Processing(SMP)上で利用され、その後、分散共有メモリ Distributed Shared Memory(DSM)システムでも利用され、より広範囲でかつ、そのスケーラビリティを大幅に向上させています。また、複数の SMP システムで構成されるクラスタシステムでは、OpenMP でのノード内の並列化とクラスタ間でのメッセージパッシングのプログラミングを併用するようなハイブリット型のプログラミングも行われています。

一般的には、共有メモリ上での並列プログラミングはメッセージパッシングによる並列プログラミングよりも容易であり、その開発工数やコストは少なくなります。共有メモリ上では、データアクセスに関して、そのデータが実際にはどのように配置されているかを考える必要はなく、また、どのようにアクセスされるかもプログラマが注意する必要はありません。これによって、プログラムの並列化の複雑化は大幅に緩和されることになります。そのため、複雑なデータ構造を持つアルゴリズムを利用したプログラムを並列化する場合や、データへのアクセスが非常に変則的な場合、また、プログラムを頻繁に変更する必要なあるような場合には、OpenMP のような共有メモリ API はメッセージパッシングでのプログラミングよりも、より容易なプログラミングや実際のプログラム開発をもたらします。

OpenMP によるプログラミングのもう一つの利点は、このプログラミング API は、他の共有メモリ上での並列プログラミング API と違って、オリジナルのプログラムに対して、指示数を挿入するだけで並列化できることです。C, C++, そして Fortran で記述した並列化されていないプログラムに並列化の指示数を挿入することで、段階的に並列化を行うことが可能であり、プログラムの一部分だけの並列化やデバックのための並列化処理の一時的な中止なども容易に制御することを可能とします。

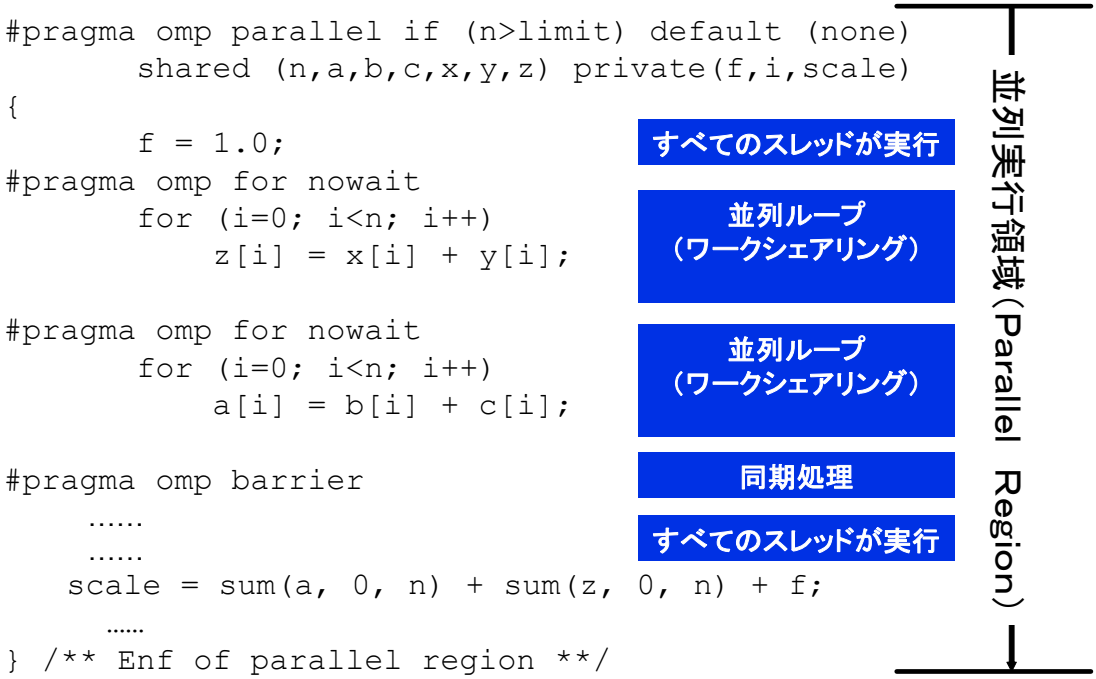

図-2 OpenMP によるプログラム構造:

OpenMP プログラミングモデル

OpenMP は、fort-join 型の並列実行を基本とします。したがって OpenMP は、最初、シングルスレッドでその実行を開始し、そのスレッドがマスタースレッドになります。

プログラム中に記述された parallel 支持数の挿入部分にプログラムの実行が進むと、その parallel 支持数が指定する並列実行領域を複数のスレッドで同時に実行します。この並列実行領域のワークシェアを図ることで、その部分をより短時間で処理することが可能となります。並列実行領域が終了すると、各スレッドは実行を終了し、マスタースレッドのみが後続の処理を行うことになります。

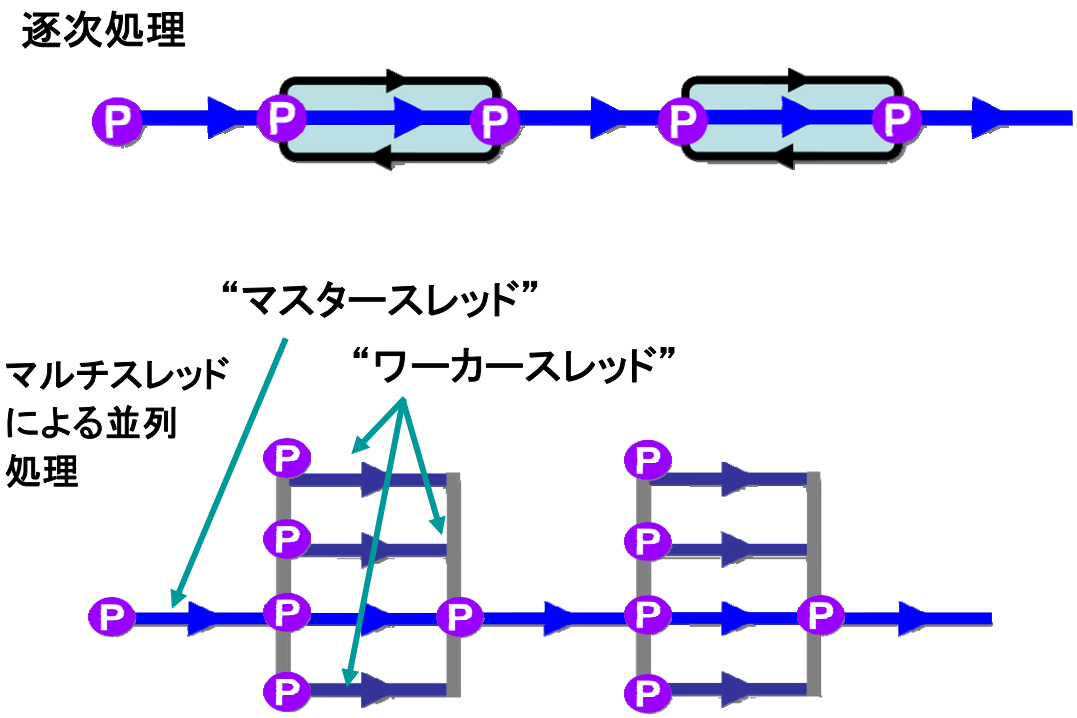

図-3 Fork-Join モデルでのプログラム実行フロー: プログラムのループなどの反復計算を複数のスレッドに分割し、並列処理を行う

この fort-join 構造は、ネストすることも可能であり、並列実行領域に parallel 指示数を指定し、さらに複数のスレッドを起動することも可能となります。

OpenMP は、同期処理のための構文が用意されています。排他実行のための critical sections や他のスレッドの実行終了を待つ barrier 同期、ある特定のオペレーションを順次実行することを指定する atomic 構文や、reduction を行うことを指定できます。また、OpenMP では、各スレッドの実行は、そのプログラムの実行順序を保証することはできませんが、その実行順序を保証することを指示することも可能です。並列プログラムで必要とされるほとんどの同期処理のための制御は、これらの指示数で可能となります。

OpenMP メモリモデル

OpenMP は、全てのスレッドによってメモリは共有されることを前提とします。全てのスレッドは直接メモリからのデータの読み込みと書き込みが可能です。OpenMP では、メモリコンシステンシは、リラックスモデルとなるため、メモリアクセスと計算のオーバラップが可能となります。

OpenMP2.5 の規定では、メモリアクセスに関して、次のように規定しています。

- メモリ領域を一次的にコピーしたローカルメモリへのアクセスもスレッドの”一時格納領域へのアクセス”として規定可能となります。
- 全ての並列実行領域での変数は並列実行領域の外側と同じ変数名となります。
- 並列実行領域を規定する parallel 指示数で共有（share）と宣言された変数は、その変数のオリジナル自身が参照されます。
- プライベート(private)と宣言された変数はオリジナル変数と同じ型と変数のサイズを持ちますが、変数は各スレッドに独自の領域に割り当てられます。

OpenMP のリラックスコンシステンシは、その実行順番に関しては非常に緩やかな規定を持つメモリコンシステンシモデルを採用しています。

- OpenMP の flush 処理はメモリの同期処理と同じになります。
- 全ての読み込みと書き込みは互いにその順番を規定されません。（もちろん、データに依存性がある場合やプログラム上の規格に従う場合は除いて）
- メモリ処理における flush オペレーションはコンパイラとシステム自身で処理される必要があります。

```
a=…..;                          （1）
＜他の仕事＞                    （3）
#pragma omp flush(a)            （2）
```

（1）メモリ内への a への書き込みは、この時点の直後から可能であり、
（2）このポイントまでには終了する必要がある

この例に示すような場合には、コンパイラは変数 a への書き込みと flash(a) の実行順序を変更することは出来ません。同時に、（1）の実行後にプログラムをマルチスレッドで処理するような場合、（2）の実行までは他のスレッドは変数 a のデータにアクセスすることは出来ません。

flash(a)の実行が終了するまで、変数 a に関するメモリ書き込みオペレーションは出来ません。

OpenMP の仕様では、変数 a に対する書き込みが完了する前でも、(3)の部分での他の計算処理の実行も可能です。実際には、(3)の部分において、変数 a の読み込みが行われてもOpenMP では、そのオペレーションを禁止しません。実際、変数 a の書き込みの後、キャッシュなどに一時保存され、メモリに書き込まれていないデータに対するアクセスも許可されています。この計算とメモリアクセスを非同期に実行することが可能なことにより、OpenMP のメモリ処理においては、メモリアクセスのレイテンシを低減することが可能となります。

Cluster OpenMP による共有メモリ並列プログラミング

OpenMP は、容易な並列プログラミングを可能とし、段階的に並列化が可能な点からも開発ワークフロー上の利点がありますが、その最大の問題は、他の共有メモリ上のプログラミング API と同じようにその実行プラットフォームがシングルアドレス空間を持つシステムに限定されることです。

共有メモリシステムは現在では、デュアル・コア・プロセッサを搭載してシングルプロセッサのシステムから数千プロセッサで拡張性を持つ大規模な NUMA システムで非常に広範囲なプラットフォームがあります。ただ、このような共有メモリシステムは一般には、千プロセッサ以上の構成では特別なインターコネクトや特別なハードウェアを必要とすることから、その価格はクラスタのようなシステムと比較すると、その価格は高くなります。また、共有メモリの実装では、例えば、共有バスを利用したシステムは、そのバスバンド幅が性能を制限することで、その性能がスケーラブルでない場合もあります。このような共有メモリシステムのスケーラビリティの問題とこのプラットフォームのコストの問題が OpenMP の普及の大きな障害となっています。

現在では、クラスタシステムが一般的となり、このようなクラスタシステム上でのメッセージパッシングプログラムは、スケーラビリティとクラスタシステム自身の価格の低下とより容易に導入運用が可能となったことにもより、並列計算のための手法として非常に成功しています。また、クラスタシステムの導入が進んでいる大きな理由の一つとしてそのコストがあります。同じプロセッサ（コア）数のシステムを単純に比較した場合、共有メモリシステムの価格は高くなります。また、システムの規模が大きくなった場合には、クラスタシステムと同程度のプロセッサ（コア）数でシステムを導入することは非常に困難となります。

インテルが開発した Cluster OpenMP は、そのようなクラスタの導入コストのメリットと OpenMP の共有メモリプログラミングの利点の双方を活用することで、並列プログラミングに新しい可能性を提供します。Cluster OpenMP という OpenMP によるプログラミングをクラスタ上にまで拡張することを可能とします。OpenMP を MPI のようなメッセージパッシングを利用することなく、クラスタ環境で利用できるようにするのが、Cluster OpenMP です。Cluster OpenMP では、陽的なメッセージ通信を行うことなく、また、ノード内とノード間のプログラミングモデルや API を変えることなくプログラミングを行うことを可能とします。

Cluster OpenMP では、通常のクラスタシステムに HW の変更を加えることなく、ソフトウェアによって分散共有メモリのサブシステムを構築し、クラスタ全体を一つの共有メモリのシステムとして、OpenMP を同様の共有メモリとしてのプログラミングを可能とします。また、Cluster

OpenMP では、OpenMP が規定するゆるやかなメモリ参照の維持によってメモリのコンシステンシが絶対的に必要な時にだけその維持を図ることを可能とします。

Cluster OpenMP は、OpenMP の容易なプログラミングをより廉価なクラスタシステムで利用可能となります。現在は、Cluster OpenMP は、Itanium ベースのプラットフォームとインテル EM64T プラットフォームで Linux オペレーティングシステム上で利用可能です。また、Cluster OpenMP は、Ethernet と Infiniband をインターコネクトとして利用するクラスタシステムについては既にその動作が検証されています。

DSM-based OpenMP プログラミング

OpenMP のリラックスメモリモデルは、分散共有メモリ(Distributed Shared-Memory:DSM)上で OpenMP を利用することを可能とします。

クラスタのリモートノードへのアクセスに関するレイテンシに関して、データの書き込みと計算をオーバラップさせることで可能とし、また、ある条件の下では、リモートからのデータの読み込みで代替することを可能とします。

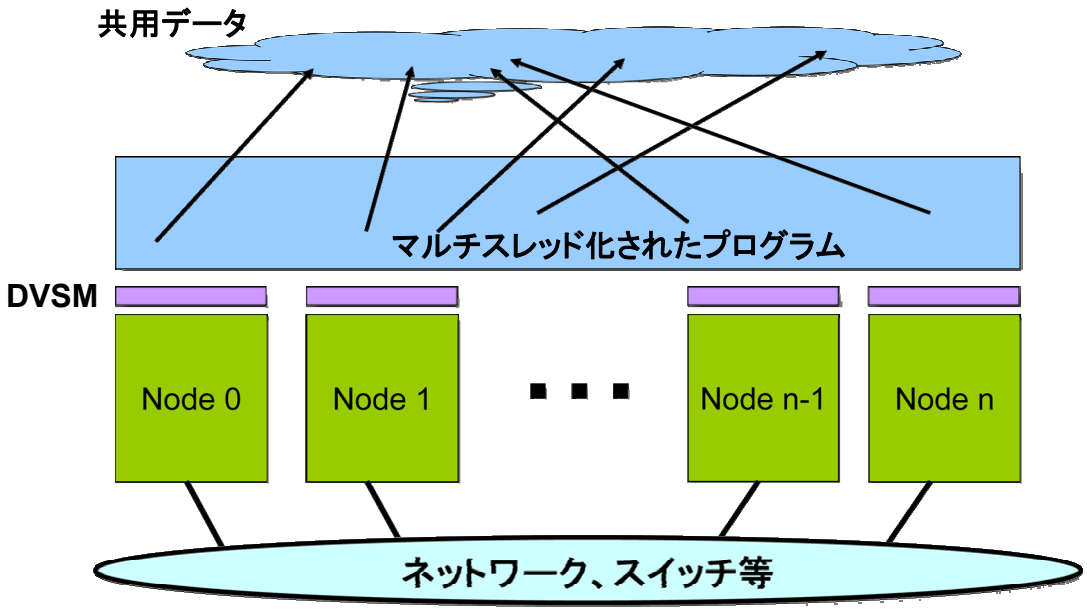

図-4 分散仮想共有メモリ(DVSM)上でのインテル クラスタ OpenMP の実装

DSM 上での OpenMP の実装

インテルは、DSM 上での OpenMP の実装を行っています。この OpenMP を Cluster OpenMP と呼び、バージョン 9.1 以降の Intel C++コンパイラと Intel Fortran コンパイラで利用

可能です。

Cluster OpenMP は、OpenMP を拡張するものであり、sharable という指示行だけを追加しています。sharable という指示行は、一つ以上のスレッドで参照される変数を指定するものです。コンパイラはプログラムのデータの配置と並列実行領域での変数の shared/private の指定に応じてコンパイラが自動でデータの sharable の指定を行います。

分散ノード間での共有変数のコンシステンシの維持は、Cluster OpenMP の実行時ライブラリが処理を行います。共有される変数は、メモリ内の特定のページにグループ化されます。このメモリページを mprotect システムコールでプロテクトすることによって実装されます。

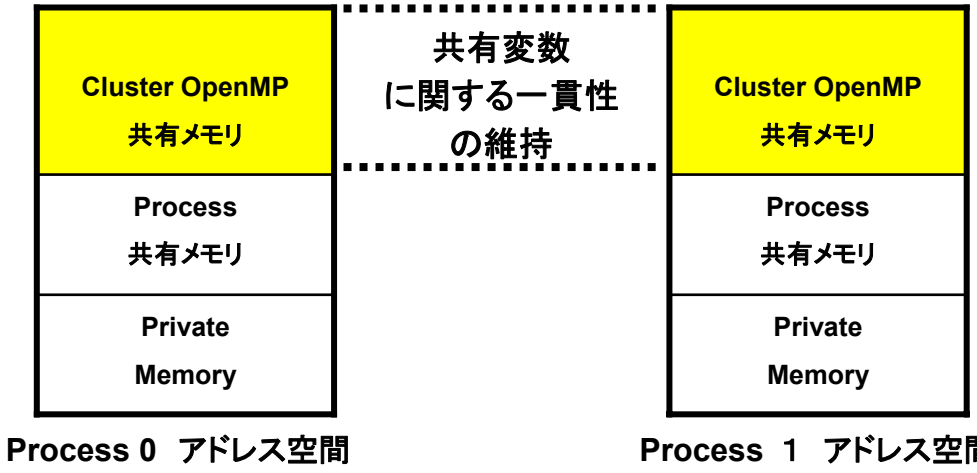

図-5 Cluster OpenMP メモリモデル:プロセス間で OpenMP スレッドがアクセスする変数は、'sharable' 変数として指示する必要があります。通常の OpenMP の共有データの宣言では、プロセス間でのデータ共有は出来ません(プロセス内でのデータの共有を宣言)

```
$ cat -n cpi.c
     1  #include <omp.h>                    // OpenMP実行時関数呼び出し
     2  #include <stdio.h>                   // のためのヘッダファイルの指定
     3  #include <time.h>
     4  static int num_steps = 1000000;
     5  double step;
     6  #pragma intel omp sharable(num_steps)
     7  #pragma intel omp sharable(step)
     8  int main ()
     9  {
    10  int i, nthreads;
    11  double start_time, stop_time;
    12  double x, pi, sum = 0.0;
    13  #pragma intel omp sharable(sum)
    14  step = 1.0/(double) num_steps;          // OpenMPサンプルプログラム：
    15  #pragma omp parallel private(x)         // 並列実行領域の設定
    16    {
    17      nthreads = omp_get_num_threads();   // 実行時関数によるスレッド数の取得
    18  #pragma omp for reduction(+:sum)        // "for" ワークシェア構文
    19      for (i=0;i< num_steps; i++){        // privateとreduction指示句
    20        x = (i+0.5)*step;                 // の指定
    21        sum = sum + 4.0/(1.0+x*x);
    22        }
    23    }
    24      pi = step * sum;
    25      printf("%5d Threads : The value of PI is %10.7f¥n",nthreads,pi);
    26  }
    27
$ icc -cluster-openmp -O -xT cpi.c
cpi.c(18) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
cpi.c(15) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
$ cat kmp_cluster.ini
--hostlist=node0,node1 --processes=2 --process_threads=2
$ ./a.out
4 Threads : The value of PI is  3.1415927
```

クラスタ間共有データの定義

コンパイルとメッセージ

並列実行処理環境の設定

図-6 Cluster OpenMP サンプルプログラムと実行例

Cluster OpenMP 利用時の注意点

メモリオペーレーションのコスト

Cluster OpenMP の利用時には、あるメモリオペレーションは他のメモリオペレーションよりもはるかにその負荷が大きくなります。良い性能を得るためには、プロテクトされていないページへのアクセスとプロテクトされているページへのアクセスの比率が非常に重要になります。これは、あるノード上でページの更新があった場合、次の同期処理の前に非常に多くのアクセスが行われることになります。

このようなことを排除するためには、同期処理を可能な限り少なくし、また、ページについては可能な限り、再利用することが必要になります。言い換えれば、atomic 構文のような細粒度での同期やロック処理は避け、高いデータの局所性を維持することが必要となります。

OpenMP メモリモデルは、メモリへの読み込みと書き込みはどのような順番でも可能です。
一方、同期処理は、その実行順番を厳密に制御する必要があります。そのため、メモリへの読み込みと書き込みは同時に実行することも可能ですが、データの flush 処理は、逐次実行が必要となり、プログラム実行時のオーバヘッドになります。

ワークロード

例えば、細粒度で並列化されたアプリケーションは、同期処理を非常に高い頻度で実行する必要があり、実行時間の大きな比重をリモートノードからのデータ取得に費やすことになります。Cluster OpenMP を利用する場合、こ並列実行の粒度が非常に重要になり、その並列実行性能を左右することになります。

例えば、非常に大規模なデータ処理を必要とするレンダリングやデータ検索、データマイニングなどの処理では一般的には、”read-only”のデータの処理の比重が非常に大きくなります。
データへのアクセスパターンは複雑で予測不可能だとしても、このような”read-only”のデータを Cluster OpenMP で処理することは容易です。”read-only”データはメモリから各ノードで読み込まれ、それぞれのノードが並列にデータ処理を行うことが可能となります。もちろん、データの read/write 処理も必要となり、その場合には処理負荷が大きくなりますが、比重的には小さなものとなります。データ構造が複雑で、データへのアクセスパターンに規則性が乏しい場合、MPI などのメッセージパッシングではその処理が複雑になるか、または全ノードが全てのデータのコピーを持つ必要があるため、その実行は現実的ではありません。

以下の例は、Cluster OpenMP を簡単な OpenMP のサンプルプログラムに適用したものです。これらのサンプルプログラムは広く利用されています。プログラムの Cluster OpenMP への変更は、非常に簡単で、sharable 宣言のための指示行の挿入だけで、クラスタシステムでの実行が可能となっています。また、そのスケーラビリティも非常に良好です。

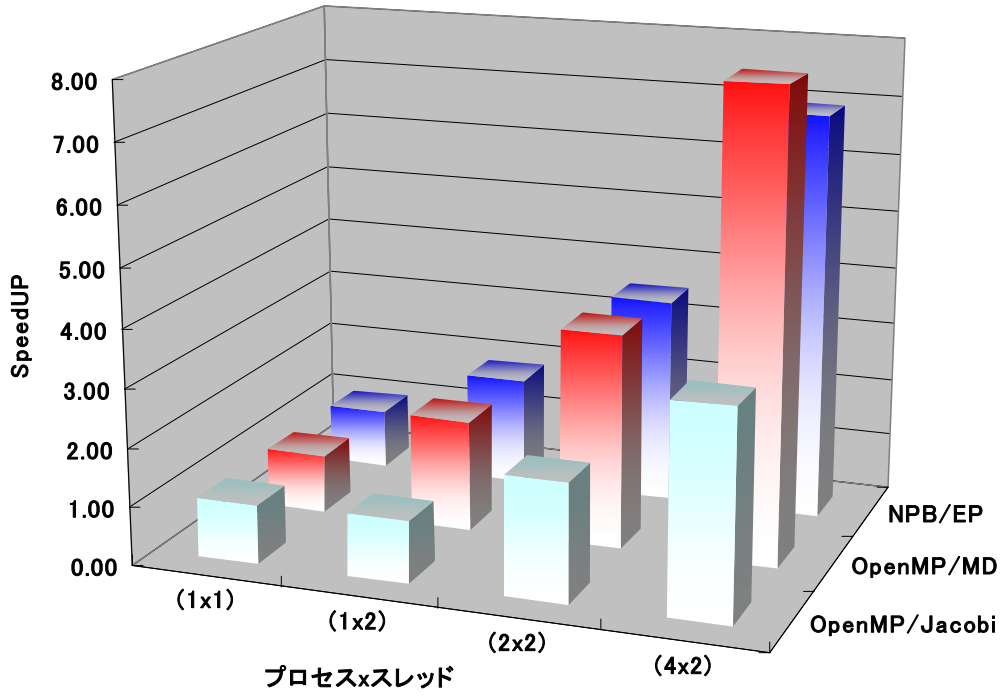

図-7 Cluster OpenMP プログラム スケーラビリティサンプル：NAS Parallel Benchmark EP, OpenMP サンプルプログラム MD(分子動力学)、OpenMP サンプルプログラム Jacobi (ヤコビ法) [1]

[1] ベンチマークシステム：NEXXUS 4820-PT/2.66GHz/1066MHz FSB/16GB Memory/InfiniBand

結論として

Cluster OpenMP を利用することで、OpenMP を利用した並列アプリケーションをクラスタ環境でより多くのプロセッサ（コア）を利用して実行することが可能となります。Cluster OpenMP は、OpenMP のリラックス・メモリ・モデルの利点を最大限に利用することで実装されており、このようなメモリ・モデルに適したアプリケーションでは、スケーラブルな性能を実現することは容易です。もちろん、全ての OpenMP アプリケーションやプログラムが Cluster OpenMP を利用して、良い性能を発揮出来るわけではありませんが、プログラムの特性を理解することでスケーラブルな性能を実現することを可能とします。同時にクラスタを利用することで、より低い導入コストでより大規模な並列アプリケーションの開発、利用環境を構築し、TCO の削減と生産性の向上を可能とします。

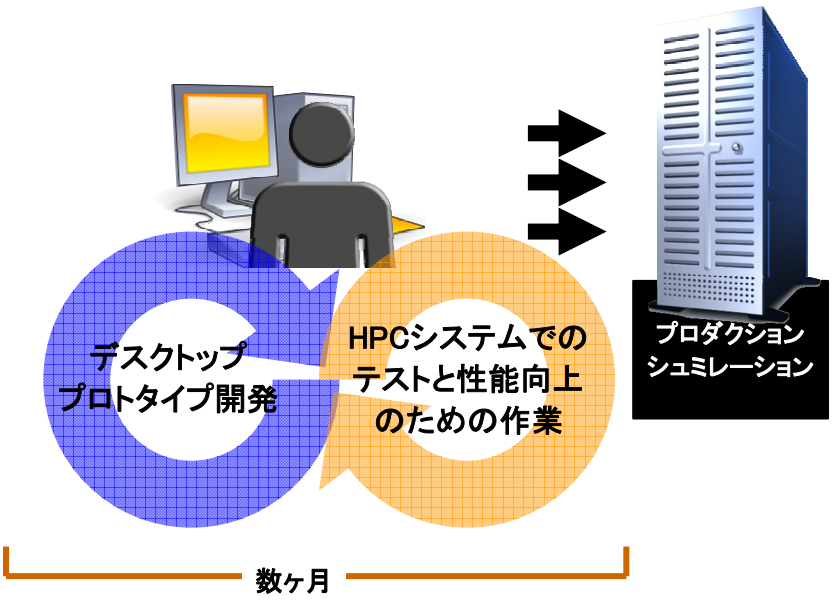

図-8 デスクトップでのプログラム開発と HPC システムでの開発を共通化することでプログラム開発のワークフローを改善し、同時にソフトウエアのギャップの克服を目指し、「Time-to-Solution」の改善を図ることを可能としてます。

参考資料

インテル® コンパイラ OpenMP* 入門
http://download.intel.co.jp/jp/business/japan/pdf/525J-001.pdf

ホワイトペーパー "Extending OpenMP* to Clusters"
http://www.intel.com/cd/software/products/ijkk/jpn/compilers/285865.htm