ソフトウェアサイエンス実験／情報システム実験／知能情報メディア実験

「高性能並列プログラミング（K-9)」実験テキスト

1. 実験の目的

本実験は，並列処理システムにおける並列プログラミングの代表的手法を学び，併せて並列処理システムの性能に関する理解を深めることを目的とする．実験では，並列プログラミング手法として，以下の 2 つを題材として取り上げる．

(1) MPI (Message Passing Interface)

MPI は分散メモリ（distributed memory）型並列計算機において，メッセージパッシングによるプロセス間通信を行うための，業界標準 API（Application Programming Interface）である．現在，ほとんどの分散メモリ型並列計算機では MPI を用いることが可能で，さらに PC（Personal Computer）やワークステーションのクラスタにおいても，標準的な並列プログラミング手法となっている．

(2) OpenMP

OpenMP は共有メモリ（shared memory）型並列計算機において，並列プロセスが 1 つの論理アドレス空間上で共有メモリを介して通信しながら並列処理を行う際，プログラム上の並列処理可能部分を簡単な記述により明示し，効率的な並列処理を行うための業界標準 API である．共有メモリ型の並列計算機のほとんどで OpenMP を用いることが可能となっており，さらに，本来は分散メモリ型であるクラスタにおいても，ソフトウェアのサポートにより仮想的な共有メモリシステムを実現し，その上で OpenMP による共有メモリ型並列処理を記述する例もある．

本実験では，数種のプログラムについて，MPI，及び OpenMP，あるいはそれらをミックスした並列プログラミングを行い，PC クラスタ上で実際にそれらのプログラムを実行し，台数効果等の性能測定を行う．さらに，得られた結果とプログラムの特性に関する考察を行い，並列処理プログラミングに対する理解を深める．

なお，MPI，及び OpenMP はいずれも，C（C++ を含む），及び Fortran の両言語に対する API が提供されているが，本実験では C（C++ を含む）のみを用いる．

2. 並列計算機アーキテクチャとプログラミングパラダイム

並列計算機あるいは並列処理システムのアーキテクチャは，プロセッサ間の通信形態に着目すると，分散メモリ型と共有メモリ型に分類できる．

分散メモリ型計算機では，各プロセッサはそれぞれ固有のメモリを持つ．すなわち，1 つのメモリシステムは 1 つのプロセッサによってのみアクセスされ，メモリの点だけを考えれば，通常の逐次計算機と同じである．並列処理においては，プロセッサ間での何らかのデータ通信が必要となるが，これはプログラム上で「通信を行う関数，またはサブルーチンの明示的な呼び出し」によって行われる．例えば，プロセッサ A がプロセッサ B にデータを与える場合，プロセッサ A 上ではそのデータの送信（send）操作を行い，プロセッサ B 上ではそれに対応する受信（receive）操作を行う．この一連の操作を**メッセージパッシング（message passing）**と呼ぶ．また，複数のプロセッサ間で各自で持つデータの総和を求めたい場合，全員がそのデータを送信し，その後でデータを纏め上げる処理が行われ，これを受信する．全体の処理の歩調は，こうしたデータの送受信の関係により自然に取られる．

一方，共有メモリ型並列計算機では，1 つのメモリシステムが複数のプロセッサによって共有され，各プロセッサは自由に任意の番地に対するデータの読み書きを行うことができる．プロセッサ A からプロセッサ B にデータを送りたい場合，特定のメモリ番地にプロセッサ A がデータを書き，その後でプロセッサ B が同じ番地からこれを読み出す．ここで重要なのは，「データが書かれた後に読む」という条件である．分散メモリにおけるメッセージパッシングでは，送信されていないデータを受信することはできないので（データがまだ到着していないため，受信プロセッサは待たされる），プロセッサ間の待ち合わせは自然に行われる．これに対し，共有メモリでは任意の番地のデータを任意のタイミングで読み出せるため，そこにあるデータが意味のあるものであることを保証する必要がある．したがって，共有メモリにおいては，何らかの形でプロセッサ同士が待ち合わせを行う必要がある．この操作を**同期（synchronization）**と呼ぶ．一般に，共有メモリプログラミングシステムでは，何らかの形でプロセッサ間の同期を取る手段が提供されている．

本実験では，分散メモリ並列プログラミングのための API として MPI を，共有メモリ並列プログラミングのための API として OpenMP を，それぞれ取り上げる．MPI，及び OpenMP の使用・記述方法・プログラム実行方法については，付録 A，及び付録 B を参照のこと．

3. PC クラスタ

近年，PC はその性能をますます増し，その対価格性能比はもはやワークステーション（WS）とは比べものにならないほど良くなっている．そこで，この PC を多数並べ，何らかの形で通信させることにより，並列処理または分散処理形態をとり，高性能計算に用いようという研究が盛んになっている．このように，多数の PC を集め，一塊にしてシステム化したものを PC クラスタ（PC cluster）と呼んでいる．

PC クラスタにおける PC をノードと呼ぶ．ノード内のプロセッサとメモリの距離が等しい SMP (Symmetric Multi Processor) と，プロセッサとメモリが対となり，プロセッサ間が接続されている NUMA (Non-Uniform Memory Access) に大別できる．本実験で用いるスーパーコンピュータ

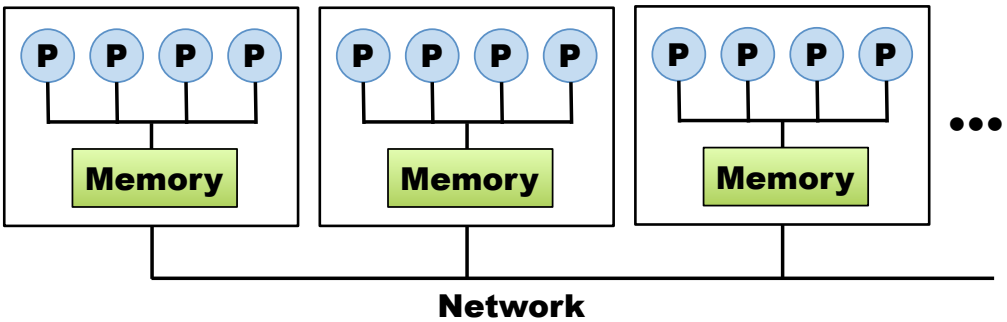

図 1　SMP-PC クラスタのイメージ図

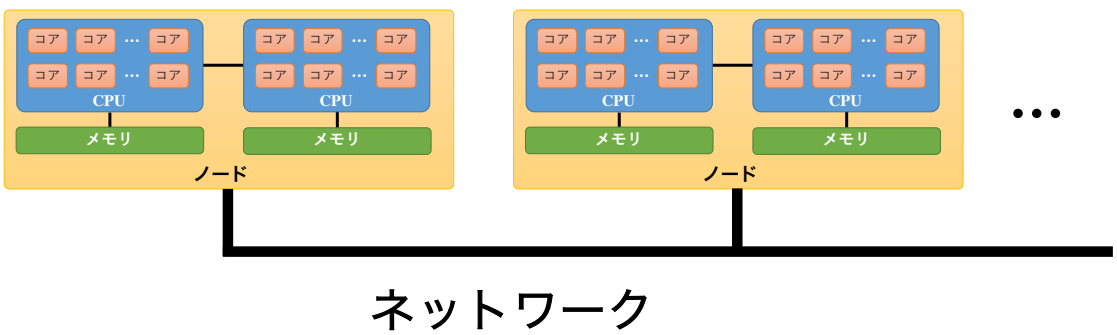

図 2　NUMA-PC クラスタのイメージ図

Cygnus は，NUMA-PC クラスタである．

PC クラスタにおけるハードウェアとソフトウェアの構成は，主に以下のような形が主流となっている．

3.1　ハードウェア

PC 自体は，ATX 等の標準的な規格の汎用品を使うことが可能である．メモリやハードディスク等についても，一般的なものである．注意すべきは PC 間を結合し，クラスタを構築するためのネットワークである．現在，1000base-T Ethernet（Gigabit Ethernet）の価格は，PC 本体に接続するカードと，ケーブルをまとめるスイッチの双方が非常に安価になっており，低価格の割に性能の高い PC クラスタを構築するための 1 つの標準となっている．この他に，高性能通信を実現するために Myrinet や InfiniBand 等を用いる場合もある．

3.2　ソフトウェア

最も重要なのは，オペレーティングシステムの選択である．これには，Windows 等ではなく，Linux を用いるのが標準となっている．Windows は PC 単体で各種アプリケーションを使うには適当であるが，ネットワーク構成を強く意識し，効率的なシステムを作ろうとする場合は不適当である（特にその不透明性が）．Linux はオープンソース UNIX であり，各種改良がユーザの手で加えられており，ネットワークの仕組みや振る舞いも明確であるため，PC クラスタでは非常に多く用

いられる．

PC クラスタにおけるノード間通信をソフトウェア的にどう見せるかについては，一般的に MPI によるメッセージパッシングが用いられる．また，各ノード上での並列プログラミングも重要になる．これには Pthread を用いる場合と，OpenMP を用いる場合がほとんどである．これらのソフトウェアは無償で提供されているものも多く，特に MPI に関しては，MPICH と呼ばれるライブラリが標準的に使われる．Pthread は Linux に標準のものが添付される．また，OpenMP についてはフリーのものがいくつか存在する．

本実験では，スーパーコンピュータ Cygnus 上で並列プログラミングを行う．MPI としては OpenMPI，または MVAPICH を用い，ノード上ではフリーの OpenMP コンパイラである GCC を用いる．Cygnus では，商用コンパイラである Intel コンパイラ (icc)，PGI コンパイラ (pgcc) も使用可能であるのでこれらを用いても構わない．

4. 並列処理と性能評価

4.1 スケーラビリティ（台数効果）

PC クラスタであれ何であれ，並列処理を行う 1 つの大きな目的は，アプリケーションの高速化である．したがって，単に物事が並列に動くだけでは（パズル的な興味は別にして）ダメで，複数のプロセッサを同時に使っただけの性能向上が求められる．この際，最も良く用いられる指標は**プロセッサ数に対するスケーラビリティ（scalability）**である．これは，**台数効果**とも呼ばれ，要するに p 台のプロセッサを用いた場合，1 台のみを用いた場合に比べ，どれくらい速くなるかということを表す．

例えば，1 プロセッサの場合の処理時間を T_1 とし，p プロセッサを用いた場合の処理時間を $T(p)$ とする．このとき，速度向上率（speed-up radio）$S(p)$ は，

$$S(p) = \frac{T_1}{T(p)} \tag{1}$$

として表すことができる．並列処理がうまくいけば，$T(p)$ は小さくなるため，$S(p)$ が大きいほど速度が向上したことになる．

4.2 並列処理効率

並列処理速度向上率 $S(p)$ は大きいほど理想的である．その目安として，$S(p)$ が p に対してどの程度大きいかに着目する．並列処理による速度向上の一つの理想は「p プロセッサを用いた場合，速度も p 倍速くなる」ということである．すなわち，

$$S(p) = p \tag{2}$$

が理想だということになる．一般には，様々な要因によってこの状況は成立しづらく，$S(p) < p$ となるのが普通である．そこで，プロセッサ数を投入したことに対する見返りがどれくらいあったかを**並列処理効率（efficiency）**として，以下のように定義できる．

$$E(p) = \frac{S(p)}{p}. \tag{3}$$

もし，$S(p) = p$ ならば $E(p) = 1$ (= 100%) であり，$S(p) < p$ であれば，1 未満となる．

4.3 並列処理効率低下の要因

並列処理効率を落とす要因は様々なものがあるが，大きく分けると，

- 並列処理可能な部分が全処理量に対し十分でない（並列度の不足）
- 並列処理をするための余分な処理時間がある（並列処理オーバヘッド）

の 2 つに分けられる．前者は，例えば仕事を p 台のプロセッサに十分分割するほど大きくない場合である．後者は，例えば仕事を並列に分割する際に通信等の余計は処理が発生することに基づく．

並列処理不可能な部分が多いと並列処理効率が上がらないということを示した，アムダールの法則（Amdahl's Law）というものがある．これは，ある処理の逐次実行時間 T_1 が，並列化できない部分 T_s と，並列化可能部分 T_p からなる場合，p プロセッサで理想的に並列化できたとしても，その並列処理時間 $T(p)$ は

$$T(p) = T_\mathrm{s} + \frac{T_\mathrm{p}}{p} \tag{4}$$

であるということを言っている．この状況で p を増やしていっても，$S(p)$ は一定値に向かって収束してしまう．

［課題］アムダールの法則に従い，式 (4) を元に，プロセッサ数 p が無限大になった極限において，$S(p)$，及び $E(p)$ がどうなるかを考察せよ．

一方，並列処理オーバヘッドは，分散メモリ方式におけるメッセージパッシング，共有メモリ方式における同期待ちなどの形で現れてくる．本来，逐次処理であれば不要であったこれらの処理は，単純にオーバヘッドとして処理時間に追加される．例えば，N 個のデータを p プロセッサに分割して処理し，最後に結果を取りまとめるような処理においては，最初に各プロセッサにデータを N/p 個ずつ分配する作業や，最後に結果を集める作業で通信が必要となる．このような通信にかかる余計なコストを**通信オーバヘッド**と呼ぶ．

4.4 メッセージパッシングにおける通信オーバヘッド

メッセージパッシングプログラムにおいては，通信オーバヘッドは比較的単純に予測可能である．モデルを単純化するために，プログラムの実行時間は，正味の計算に要する部分 T_calc と，通信を行う部分 T_comm に分かれると考える．総実行時間は，$T_\mathrm{calc} + T_\mathrm{comm}$ として表される．

通信時間は，一般的に，通信するデータの量に依存する．ここで注意しなければいけないのは，この関係は**比例関係ではない**，ということである．通常，通信時間は**通信データ量に依存しない固定コスト**と，**データ量に比例するコスト**の和で成り立つ．すなわち，データの量を N とすると，通信時間 T_comm は

$$T_\mathrm{comm} = a + \frac{N}{b} \tag{5}$$

で表される．ここで，a は固定コスト，b は単位時間あたりに転送できる通信量単位である．

［課題］スーパーコンピュータ Cygnus における，ノード間 MPI 通信の性能を調べ，式 (5) の a, 及び b に相当するパラメータを求めよ．同様に，ノード内のプロセッサ間で MPI 通信を行った場合についても調べよ．

一般的に，メッセージパッシングによる並列プログラムでは，各プロセスは何らかの内部処理（演算）を行った後，通信を行う．その後さらに内部処理を行い，次に通信，$\cdots$ というように，内部処理と通信を交互に繰り返す．当然，内部処理に要する時間が長く，通信データ量が少ないほど，通信オーバヘッドは軽くなる．そこで，この両者の比を概念的に**並列処理粒度（granularity）**と呼ぶ．内部演算が長く，通信時間が短い場合を**粒度が粗い（coarse grain**）と呼び，内部演算が短く，通信時間が長い場合を**粒度が細かい（fine grain）**と呼ぶ．

4.5 共有メモリプログラムにおけるオーバヘッド

共有メモリパラダイムの場合，通信のオーバヘッドはメッセージパッシングの場合と違って目に見える形では現れにくい．しかし，共有データの参照のために排他制御を行ったり，同期待ちを行う処理は明らかにオーバヘッドである．

また，OpenMP の場合，並列化を行う際に並列プロセス（スレッド）を起動したり停止したりする必要がある．さらに，それらが並列動作中に，排他制御を必要とする場合，その制御にも時間がかかる．その他，private 属性を持つ変数を大量に使うような場合（付録 A 参照）も並列プロセスの起動に時間がかかることになる．

5. 実験内容

5.1 OpenMP による並列化実験

以下の各種処理を OpenMP によって並列化し，1 つのノード上の複数のコアで性能評価せよ．まず，付録 A の「OpenMP チュートリアル」にある例（hello world, laplace 等）を実験してみよ．その後で，以下の例題を自分で記述し，実行してみよ．

(a) 配列データの総和を求めよ．（整数，浮動小数）
(b) 配列データに適当な重みの関数を作用させ総和を求める．例えば，$1/m \sin(1/m)$ を多数の $m\,(= 1, 2, \ldots, N)$ に対して計算し，総和を求めてみよ．
(c) m 行 k 列の行列 A と，k 行 n 列の行列 B の積を計算し，m 行 n 列の行列 C を求めよ．

5.2 MPI による通信実験

付録 B「MPI による並列プログラミング」に従い，以下の各通信実験を MPI によって行い，通信性能を比較せよ．

(a) ノード間の point-to-point 通信性能を各種データ量について測定せよ．
(b) ノード内の同一 CPU 上のプロセス間での point-to-point 通信性能を測定せよ．

(c) ノード内の異なる CPU 上のプロセス間での point-to-point 通信性能を測定せよ．

(d) 各種データサイズに対し，collective 通信（allgather 等）の性能を測定せよ．

(e) point-to-point 通信を 2 者間のみで行う場合と，そういったペアの通信を多数同時に行った場合について性能を比較せよ．

5.3 OpenMP，及び MPI を独立に用いた応用プログラミング

N 点の質点間の重力相互作用に基づき，ニュートンの運動方程式の積分を行うプログラムを，OpenMP，及び MPI でそれぞれ別個に作成せよ．

質点 i と質点 j の間には，互いに以下のような x, y, z 方向の加速度 $a_{x_{i,j}}, a_{y_{i,j}}, a_{z_{i,j}}$ が存在する．ここで，G は万有引力定数，m_j は質点 j の質量，$r_{i,j}$ は質点 i と質点 j 間の距離であり，x_i, y_i, z_i はそれぞれ，質点 i の x, y, z 方向の座標を表す．

$$
\begin{aligned}
a_{x_{i,j}} &= G \cdot \frac{m_j}{r_{i,j}^2} \cdot \frac{x_j - x_i}{r_{i,j}}, \\
a_{y_{i,j}} &= G \cdot \frac{m_j}{r_{i,j}^2} \cdot \frac{y_j - y_i}{r_{i,j}}, \\
a_{z_{i,j}} &= G \cdot \frac{m_j}{r_{i,j}^2} \cdot \frac{z_j - z_i}{r_{i,j}}.
\end{aligned}
\tag{6}
$$

これらにより，質点 i と質点 j の関係における x, y, z 方向の加速度が求まる．

次に，各質点の速度を求める．現在のタイムステップにおける質点 i の x, y, z 方向の速度をそれぞれ，$v_{x_i}, v_{y_i}, v_{z_i}$ とする．また，質点 i の次のタイムステップにおける x, y, z 方向の速度をそれぞれ，$v_{x_i}^{(\mathrm{next})}, v_{y_i}^{(\mathrm{next})}, v_{z_i}^{(\mathrm{next})}$ とすると，これらは以下の式で求められる．

$$
\begin{aligned}
v_{x_i}^{(\mathrm{next})} &= v_{x_i} + \Delta t \cdot \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{x_{i,j}}, \\
v_{y_i}^{(\mathrm{next})} &= v_{y_i} + \Delta t \cdot \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{y_{i,j}}, \\
v_{z_i}^{(\mathrm{next})} &= v_{z_i} + \Delta t \cdot \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{z_{i,j}}.
\end{aligned}
\tag{7}
$$

ここで，Δt は時間の刻み幅に相当する定数である．

最後に，質点 i の座標を求める．質点 i の現在のタイムステップにおける x, y, z 方向の座標をそれぞれ，x_i, y_i, z_i とする．質点 i の次のタイムステップにおける x, y, z 方向の座標 $x_i^{(\mathrm{next})}, y_i^{(\mathrm{next})}$, $z_i^{(\mathrm{next})}$ は，以下で求められる．

$$
\begin{aligned}
x_i^{(\mathrm{next})} &= x_i + \Delta t \cdot v_{x_i}^{(\mathrm{next})}, \\
y_i^{(\mathrm{next})} &= y_i + \Delta t \cdot v_{y_i}^{(\mathrm{next})}, \\
z_i^{(\mathrm{next})} &= z_i + \Delta t \cdot v_{z_i}^{(\mathrm{next})}.
\end{aligned}
\tag{8}
$$

各質点の初期位置は，3 次元空間上でランダムに与えることとし，各質点の質量 m_i $(i = 1, 2, \ldots, N)$ は 1 から 2 の間でランダムに与えるものとする．

5.4 重力プログラムのハイブリッド化

MPI と OpenMP の両方を用いた形で，重力プログラムをハイブリッド化せよ．ここで，MPI と OpenMP のハイブリッドプログラムとは，以下のようなものを指す．実験で用いる Cygnus では，OpenMP のプログラムは同一ノード上の CPU 間でしか有効でない．これに対し，MPI プログラムは hostfile をどのように設定するかによって，並列プロセスをどの CPU，及びコアに割り当てるかが決まる．Cygnus では hostfile は自動的に生成されるため，自分で設定する必要はない．また，Cygnus では

- 各ノード上に 1 つずつしか MPI プロセスを置かない場合（全プロセス数は，ノード数と等しい）
- 各 CPU に 1 つ MPI プロセスを置く場合（全プロセス数は，ノード数 $\times 2$）
- 各コアに MPI プロセスを置く場合（全プロセス数は，コア数と等しい）

といった割り当てが考えられる．この割り当ての指定は，OpenMPI の引数によって指定することができる．同一ノード内に複数の MPI プロセスが割り当てられた場合は，ノード内の複数のコア，CPU 間でも MPI による通信が行われる．

これに対し，ハイブリッドプログラミングでは，まず各ノードに 1 プロセス，ないし複数の MPI 並列プロセスを立ち上げる．そして，各プロセス内で適宜 OpenMP 記述を行うことにより，ノード内でも部分的な並列処理を行う．こうすると，ノード内では OpenMP による共有メモリ並列処理が，ノード間では MPI による分散メモリ並列処理が行われることになる．ノード内での CPU 間通信は，MPI 関数を用いてメッセージパッシングを行うより，共有メモリを直接読み書きした方が速いと考えられる．すなわち，ハイブリッドプログラミングによって，ノード内通信が最適化される可能性があり，MPI だけを用いた場合より性能が上がる可能性がある．

しかし，実際にはハイブリッドプログラミングは複雑であり，また種々の要因によって，必ずしも MPI のみを使う場合よりも高速化されるとは限らない．そこで，ハイブリッドプログラミングを実際に行い，以下の組み合わせについて性能を比較せよ．比較は様々な角度から行い，自分なりに結論をまとめること．

- 1 ノードを用いた場合
  - 24 MPI プロセス × 1 OpenMP スレッド実行
  - 2 MPI プロセス × 12 OpenMP スレッド実行
  - 1 MPI プロセス × 24 OpenMP スレッド実行
- 2 ノードを用いた場合
  - 48 MPI プロセス × 1 OpenMP スレッド実行
  - 4 MPI プロセス × 12 OpenMP スレッド実行
  - 2 MPI プロセス × 24 OpenMP スレッド実行
- 4 ノードを用いた場合
  - 96 MPI プロセス × 1 OpenMP スレッド実行
  - 8 MPI プロセス × 12 OpenMP スレッド実行
  - 4 MPI プロセス × 24 OpenMP スレッド実行
- 8 ノードを用いた場合
  - 192 MPI プロセス × 1 OpenMP スレッド実行
  - 16 MPI プロセス × 12 OpenMP スレッド実行
  - 8 MPI プロセス × 24 OpenMP スレッド実行

図 3　MPI プロセス数と OpenMP スレッド数の組み合わせ

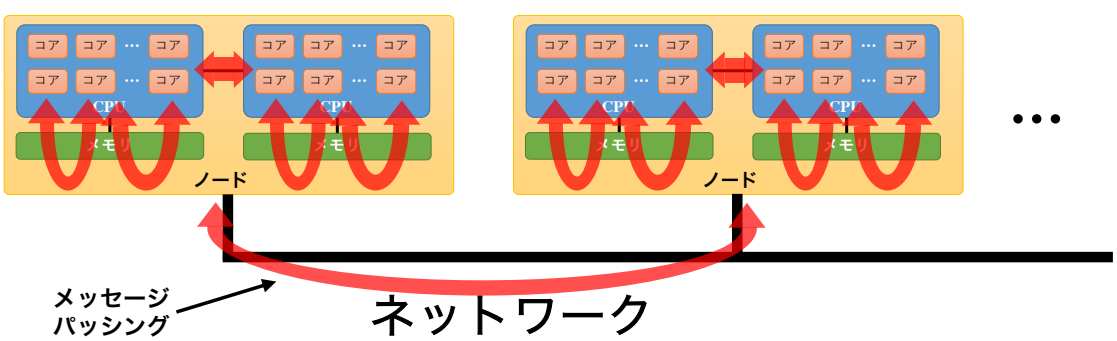

図 4　MPI のみを用いた場合の通信

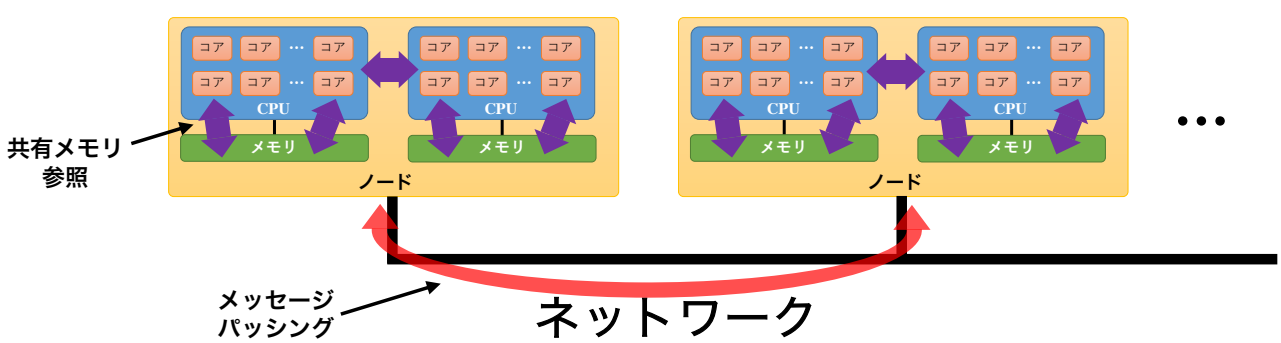

図 5　MPI+OpenMP ハイブリッドプログラミングの場合の通信（1 ノード 1MPI プロセスの例）