５．クラスタ・NORA(NORMA)

並列コンピュータを作る最も簡単な方法は、コンピュータ同士を何らかのネットワークで接続して、データを交換できるようにすることだ。Ethernet で接続されている複数のコンピュータは、それが NFS でファイル共有しているだけならば並列コンピュータとは言えないが、何等かの形でプログラム同士がデータ交換をしながら走ることができればそれは並列コンピュータと言える。このために数値計算用には PVM（Parallel Virtual Machine）や MPI（Message Passing Interface）などのメッセージパッシング用のライブラリが用意されている。ビッグデータを処理用するデータセンターでは、基本的には独立した要求を処理すれば良いが、単独の要求に対しても MapReduce などのプログラミング手法で並列検索を行う。これらの共有メモリを持たない並列コンピュータは、クラスタあるいは NORA・NORMA（No-remote memory access model）と呼ばれる。

クラスタは、高速なネットワークでコンピュータを接続すれば良いので、最も低コストで大規模システムを作ることができる。クラスタ型コンピュータの鍵を握るのは、コンピュータ間を接続する結合網で、多くのクラスタでは、Infiniband と呼ばれる System Area Network(SAN)と呼ばれる低レイテンシィの専用結合網を用いるか、40G/100G の高速 Ethernet を用いている。最近は、これらのネットワークを用いて、他入力、他出力のスイッチにより高バンド幅のネットワークを構築する手法（ハイラディックスネットワーク）が用いられる。これらの結合網については、6 章に紹介することにし、ここではいくつかクラスタ型のアーキテクチャに触れて、メッセージパッシング用ライブラリとして MPI を紹介する。

5.1 ベオウルフ型クラスタ（PC クラスタ）

1990 年代、専用のプロセッサと専用のネットワークを用いたスーパーコンピュータが高性能計算の主流を占めていた。ベオウルフ型クラスタはこれに対抗して 1994 年に NASA で始まったプロジェクトで、安価で簡単に大規模な並列計算環境を作るために以下の 4 点を方針として掲げた。

1. コモディティの PC を利用する
2. コモディティのネットワーク（Ethernet）を利用する。
3. コモディティのソフトウェア（Linux）を利用する。
4. プログラムは PVM、MPI などのメッセージパッシングライブラリにより行う。

この方式はメッセージ交換のプログラムをプログラマが書かなければならない点で負担は大きく、ネットワーク遅延による性能低下は避けられないが、専用のプロセッサと専用のネットワークを持つ同規模の並列コンピュータに比べてはるかに安価であり、同程度の予算ならはるかに規模を大きくすることができた。以降、この方式は、安価にスーパーコンピュータを実現する方法として広がった。現在も、コモディティの GPU をコモディティのネットワークで接続したスーパーコンピュータは、この系譜を引き継ぐと言える。

図 1 は筆者らが開発したクラスタ RHiNET-2 で、ベオウルフ型の特徴１．３．４を持って

いるが、ネットワークは光を用いた強力なものを利用していた。

5.2　ウェアハウススケールコンピュータ（WSC）

Web サービス、データベース処理、トランザクション処理を行うデータセンターは、当初は今まで紹介した分散共有メモリ型の並列コンピュータや、ベオウルフ型のクラスタの集合体で構成されていた。しかしクラウドコンピューティングの発達と共に、Google、Amazon などのデータセンターは巨大化して新しいコンピュータとしての性質を帯びるに至った。データセンターでは、基本的には別々の要求を並列処理するので、処理しているノード間に直接のデータ交換は必要ないが、巨大なデータと大規模なネットワーク、巨大な電源、冷却装置を共有する。さらに検索処理を代表とするビッグデータ処理では単独の要求に対しても多数のノードがデータ検索を並列に行う枠組みが用意されるようになった。

データセンターが巨大になり、その数を増やすと、コモディティの PC を利用する必要がなくなった。それ自体で十分な数を必要とするため、データセンター専用のプロセッサを開発しても十分引き合うようになったのだ。管理上、様々なシステムが同居することは許されないため、同じ構成を全システムが使うようになった。

1. 専用のプロセッサを全てのシステムで用いる
2. ソフトウェアに頼って信頼性を維持し、故障ノードはどんどん交換する。
3. ネットワークは遅延よりもバンド幅重視
4. プログラムは MapReduce などビッグデータ処理用の言語を用いる

Hennessy と Patterson は有名な Computer Architecture – a quantitative approach の 5 版からこのコンピュータを Warehouse Scale Computer（WSC）という新しいクラスであると主張した。WSC はまださほど広く使われる言葉とはなっておらず、巨大データセンターを支える技術は並列コンピュータアーキテクチャ以外の所に依存するところが大きいが、その重要性は大きいため、知っておくことは必要である。

5.3 メッセージパッシングライブラリ

並列プログラミングは共有メモリを利用する方法と、メッセージパッシングを利用する方法に分類される。メッセージパッシングは、共有メモリを持たない並列コンピュータがやむを得ず用いる方法ではなく、形式的検証がしやすく信頼性の高いプログラムが可能であるという大きな利点を持っている。これは、共有メモリが変数を介してデータをやり取りするため、どこでだれが書いてだれが読むのかを明確にすることが難しいためだ。読み書きの順番も前章に示したように自由度が高いので、これを検証することは不可能に近い。これに対してメッセージパッシングは、どこでどのプロセッサ間でどのようなデータが交換されるのかがプログラマにより明示されているため、分かりやすい。また、メッセージの待ち合わせにより同期が行われるため、同期の生じる場所、原因、それに伴うデータ交換が明らかである。このため、メッセージパッシングを利用した言語に対するプログラムの正当性を証

明する方法は長い間研究されている。ではまずこの通信の方法を簡単に紹介する。

5.3.1 ブロッキング通信とノンブロッキング通信

送信側のプロセッサ（プロセス）は、受信側を指定して send 関数を実行し、受信側のプロセッサは送信側を指定して receive 関数を実行する。実行時に、相手方が対応する関数を実行していれば、データは転送され、両者は処理を続ける。この様子を図 2 に示す。どちらが先でも、先に実行した方は、相手が実行するまで待ち続ける。この方式をブロッキング通信あるいはランデブと呼ぶ。この方法は待ち時間が長く、性能が低くなるが最も安全で確実な方法である。

メッセージのバッファを設けることで、送信側は先に進むことができるだろう。send を実行した送信側は受信側が待っていなければ、受信側のメッセージバッファにメッセージを入れて先に進む。受信側は、receive 実行時に、バッファにメッセージが存在すれば、それを受け取り、先に進む。そうでなければ待ち状態になる。この方法は簡単だが、受信側は先に着いた際に待たされるので不公平な感じがする。このため、receive で待ち状態になった受信プロセッサは別なプロセスに切り替えて、待っている間を有効利用する。この枠組みがノンブロッキング通信であり、ブロッキング通信に比べて性能面では高いがプログラムが複雑になる。本書ではブロッキング通信のみ扱うことにする。

5.3.2 MPI (Message Passing Interface)

MPI はメッセージパッシングライブラリで OpenMP 同様 C や Fortran のプログラムから呼び出して利用する。先行して普及したライブラリ PVM（Parallel Virtual Machine）の上位互換であり、グループ通信やタグを使ったエラー処理など多様な機能を持っている。OpenMP と違って pragma はなく、全てを関数呼び出しの形で行う。プログラムは SPMD（Single Program Multiple Data streams）の形を取る。SPMD とは同じプログラムで多数のデータを一括して処理する方法を指す。SIMD（Single Instruction Multiple Data streams）よりもプログラム上の制約が緩く、基本的に流れが同じプログラムをプロセス id により切り替える。すなわち、id に応じて行う処理を分けることができる。基本的には OpenMP も SPMD と言えるが、MPI の場合共有メモリがないため、id によって送信側と受信側を分けることになり、プログラムはやや複雑になる。ここでは、MPI の中で基本的な関数のみを紹介する。興味を持ってさらに習得したい読者は文献［］などを参照されたい。

MPI は以下の 6 つの関数が基本である。

① MPI_Init(int *argc, char ***argv);　MPI の初期化関数でこれを実行しないと他の関数が使えない。　引数は main 関数の引数を以下のように使うのが普通

```
int main(int argc, char *argv)
  MPI_Init(&argc, &argv);
```

これにより main から MPI に設定を行うことができるのだがここでは気にしないことにす

る。

② MPI_Comm_rank(MPI_Comm, int *rank); プロセス(プロセッサ)id を取ってくる関数。MPI の用語ではプロセス id のことを rank と呼ぶ。第一引数の MPI_Comm は Communicator といって MPI で通信する空間を定義する。ここに MPI_COMM_WORLD を用いると、全プロセッサが通信する空間を指す。
MPI_Comm_rank(MPI_COMM_WORLD, &id); を実行するとプロセス id を取ってくることができる。

③ MPI_Comm_size(MPI_Comm, int *size); 全プロセス数を返す。
MPI_Comm_size(MPI_COMM_WORLD, &nproc); を実行すると、全プロセス数 nproc を取ってくることができる。

④ MPI_Send　1 対 1 のブロックメッセージ通信。引数が以下のようにたくさんある。
MPI_Send(
void *buf,　　送信用バッファ
int count,　　送信する要素数
MPI_Datatype　MPI のデータタイプ
int dest,　　受信側のプロセス id
int tag,　　通信時のタグ　ここでは使わない
MPI_Comm　MPI Communicator で MPI_COMM_WORLD を指定
);
MPI_Send(msg, MSIZE,MPI_CHAR,0,0,MPI_COMM_WORLD);　は
転送用の文字列配列 msg 中の文字列を MSIZE 分プロセス 0 に送る（タグも 0）という意味になる。ここで、MPI のデータタイプは、基本的に通常のデータ型と同じで、MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE などが定義されている。

⑤ MPI_Recv　1 対 1 のブロックメッセージ受信で Send_recv に対応する。
MPI_Recv(
void *buf ,　受信用バッファ
int count,　受信する要素数
MPI_Datatype　　MPI のデータタイプ
int source,　　送信側のプロセス id
int tag　　送信メッセージタグ
MPI_Comm,　MPI Communicator で MPI_COMM_WORLD を指定
MPI_Status　　受信したメッセージの状態
)；
MPI_Recv(msg, MSIZE, MPI_CHAR, 1,0, MPI_COMM_WORLD, &status);　は、先の MPI_Send に対応する受信を行う。タグとプロセス id が合わないとメッセージは受信されない。

⑥ MPI_Finalize(); MPI の終了

例題として有名な Hello World をプリントする hello.c を図 3 に示す。まず MPI ライブラリを用いる場合には<mpi.h>を指定する必要がある。ここではメッセージ転送用のバッファ msg を用意する、メッセージサイズは 64 にしておく。MPI は MPSD の考え方に基づくので、MPI__Init をした時からそれぞれのプログラムは並列に走ると考える。まずプロセス番号 pid とプロセス数 nprocs を取ってくる。pid==0 がホストの役割を果たす。ホストはそれ以外のプロセスからメッセージを受け取り、その都度出力する。一方、その他のプロセスは、Hello world!という文字列をメッセージバッファ msg に入れ（sprintf)、これをホストに対して転送する。

MPI のプログラムのコンパイルは以下のように行う
mpicc -o hello hello.c
実行時に、-np の後にプロセッサ数を指定する。
mpirun -np 4 ./hello
各プロセッサからの Hello world!が表示されるはずである。

例題のリダクション演算と演習をやってみよう。