Preprint published as: A. SaiToh, Building a small-scale cluster machine—a hands-on report, Bulletin of Sojo University **45**, 75-96 (2020), in Japanese.
プレプリント 出版情報：齋藤暁, 小規模クラスタマシン制作報告, 崇城大学紀要 第 45 巻, 75-96 (2020).

崇城大学　紀要
第 45 巻　令和 2 年 3 月

小規模クラスタマシン制作報告

齋藤 暁*

Building a Small-Scale Cluster Machine – a Hands-on Report

by
Akira SAITOH *

要 旨

科学技術計算を行うための汎用的かつ自由度の高い計算資源は、シミュレーション研究の基盤である。今回、情報学科のサーバ室に、一般的な CPU と GPU および 10Gbit NIC を搭載した PC 7 台からなるクラスタマシンを構築したので、構築過程とベンチマークテスト結果を報告する。ソフトウェア環境に関しては比較的最近の Linux OS である CentOS 7 を使用して徹底して標準的な並列計算用の環境構築に努めた。Linpack によるベンチマークでは機材構成からして妥当な演算性能を示した。

Key Words：並列計算、クラスタマシン、ＭＰＩ、ベンチマーク

1. はじめに

昨今、CPU のメニーコア化が進み、PC 向け単一 CPU でも普及価格帯で 8～16 コア、[1] ワークステーション向けで24～64 コアが使われるようになってきた [2] が、単一の計算ノードでは大規模な科学技術計算用の資源を賄うには至っておらず、まだまだクラスタマシンとスーパーコンピュータの需要は旺盛である [3]。

本学には大規模なクラスタマシンやスーパーコンピュータは設置されておらず、計算資源が必要であれば九州大学の ITO や大阪大学の OCTOPUS、東京工業大学の TSUBAME といった全国共同利用施設のスーパーコンピュータを利用することになる。その際、計算ジョブを投入する前に複数ノードで正しく動作するか、計算速度のスケーリングが見込めるか、予め確認しておくのが通常の利用者の心得である。これは共同利用の計算資源を無駄に使わないためでもあるが、有り体に言えば使用できる CPU 時間が料金によって決まっているためである。他大学の事例では科学技術計算を行う研究室や講座では自前のクラスタマシンをテストベッドとして構築していることが多い [4]。物理クラスタマシンではなく仮想クラスタマシンをクラウド計算基盤上に構築して使うというアプローチもないではないが、仮想化されたネットワーク [5] およびメモリ等 [6] の性能低下が看過できない。

もちろん、単にテストベッドとしてではなく中規模の科学技術計算のためにも自前のクラスタマシンは気兼ねなく使えるため好都合である。最近では理化学研究所の京の次世代機へのリプレースに伴う運用停止もあってスーパーコンピュータは全般的に利用が混み合っており、私の経験では ITO の共用ノードではジョブ投入後の実行

＊崇城大学情報学部情報学科准教授

が 24 時間以上後、という場合がしばしばある。

こういった背景の下、今回 7 ノードのクラスタマシンを構築して情報学科サーバ室に設置した。構成要素はハードウェア、ソフトウェア共に標準的なものを揃えた。特に OS と並列プログラミング用のライブラリについてはスーパーコンピュータと環境の差異がなるべく小さくなるように心掛けている。なお、同規模のクラスタマシン構築事例としては文献 [7] が参考になる。

本稿ではまた、構築したクラスタマシンの性能を High Performance Linpack (HPL) [8] を用いて評価し、単一ノードのワークステーションおよび公表されている既存の他システムと実行効率を比較する。

以下ではまず 2 節でクラスタマシンの構成を紹介し、3 節で実際の構築過程の詳細を述べる。4 節で性能評価を行い、5 節で議論、6 節でまとめを述べる。

2．クラスタマシンの構成

本節ではハードウェア構成とソフトウェア環境について述べていく。まず、ハードウェアはもっぱら民生用の安価な量販機材を用い、クラスタ内部の接続には 10Gbit イーサネットを使用して構築した。構築後情報学科サーバ室で稼働しているところを図-1 に写真で示す。また、クラスタ内部および外部とのネットワーク接続の略図を図-2 に示す。

計算を走らせるのは図中の node0～node6 の計算ノードであり、node0 は管理ノードを兼ねている。各ノードは同一構成の PC であり、構成は表- 1 に示すとおりである。計算ノード間の通信は通常並列計算のボトルネックであるので、可能な限り高速なネットワーク機器を用いるべきだが、予算の都合上 2018 年中期時点でボード 1 枚あたり数万円単位で揃えられる 10Gbit イーサネットを採用した（表-1)。参考までに、10Gbps を超える通信速度の機材（QDR InfiniBand や 25GbE、40GbE）は価格が 1 桁高くなる。IP アドレスは、クラスタ内部は VPN ルータの DHCP サーバ機能を使って、node0～node6 の起動時に NIC の MAC アドレスに対して固定 IP アドレス

図-1　クラスタマシン外観

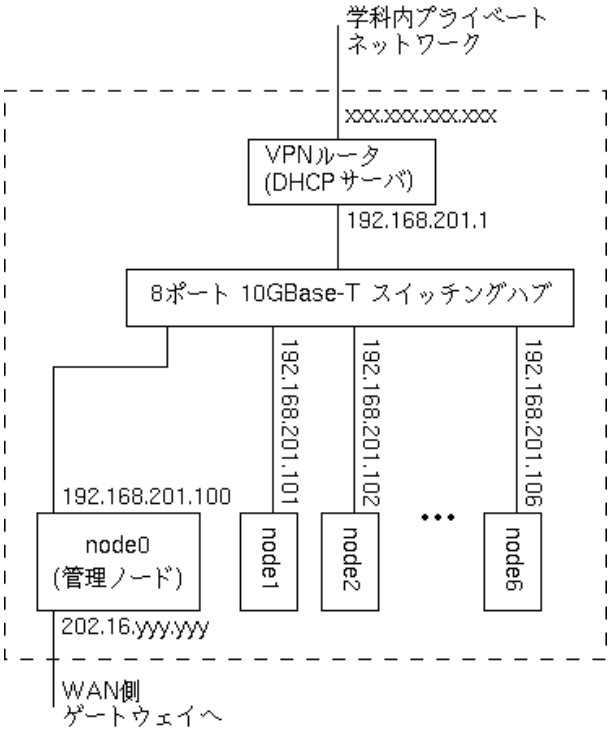

図-2　ネットワーク接続図。点線の枠内がクラスタ内部である。なお、図中の各機材については本文と表-1 を参照のこと。

表-1　構成機材のリスト

図-2 での表示	詳細
node0～node6	市販 PC にメモリと拡張ボードを増設したもの。 CPU: Intel Core i7-8700K (6 cores, 12 threads, 3.7-4.7GHz) メモリ: DDR4-2666　32GB GPU: NVIDIA GeForce GTX 1060 3GB GDDR5 NIC: Aqrate AQR105 10GBase-T カード& Intel 1GbE オンボード ストレージ: node0 は 6TB HDDx2 (RAID1)、node1～6 は 1TB HDD UPS: オムロン BW55T その他：DVD-RW ドライブ 他
8 ポート 10GBase-T スイッチングハブ	NETGEAR XS508M-100AJS 8 ポート 10GBase-T スイッチングハブ
VPN ルータ	TP-Link TL-R600VPN

192.168.201.100～192.168.201.106 を割り当てている。ノード間通信は起動後はスイッチングハブを介するのみであるので、VPN ルータは 1Gbps の機材（表-1）を使用しているが並列計算時の通信速度は 10Gbps である。

クラスタ外部との接続については、2 系統あり、まず学科内のプライベートネットワークに VPN ルータを介して各計算ノードをつなげている。これは、主に各計算ノードの OS とソフトウェアのアップデートをする際に、学内にあるミラーサイトへアクセスするためと、学外のレポジトリに学内のプロキシサーバを経由してアクセスするために用いる。もう一系統は、学内のグローバルアドレスセグメントの建屋内ゲートウェイと node0 を接続している。これは利用者がアクセスするために用いる。今のところは大学の基幹ファイアウォールで学外からのアクセスは遮断しているが、将来的にはこの遮断を解除すれば学外からのアクセスが可能である。なお、node0 のグローバル IP アドレスは学内のネームサーバでホスト名 cluster.sncq.cis.sojo-u.ac.jp と対応づけられている。クラスタ外部との接続は建屋のネットワーク機材に制限せれて、いずれも 100Base-TX を使用しているが、もちろんクラスタ内部の通信速度には一切影響しない。

続いてソフトウェア環境について述べる。OS は CentOS 7.4 を、SELinux を disable、firewalld を enable した状態で使用した。ユーザ認証は Network Information Service (NIS)を使用した。node0 で NIS サーバと NIS クライアント、node1～node6 で NIS クライアントを走らせる。同時に、ユーザのホームディレクトリは Network File System (NFS)で node0 上のものをマウントして使う。node0 で NFS サーバ、node1～node6 で NFS クライアントを走らせる。数値計算用のライブラリとしては、GMP 6.0.0、MPFR 3.1.1 等をインストールし、GPGPU 用のライブラリは CUDA 9.1 をインストールした。並列計算用の通信方式としては標準的な Message Passing Interface (MPI)を利用することにし、MPI 準拠の実装として普及している MPICH の ver.3.2 をインストールした。これらを含むソフトウェア環境の構築の詳細は、次節を参照されたい。表-2 に導入したソフトウェア

表-2 導入ソフトウェアのリスト（一部）

名称または略称	詳細
CentOS	ver.7.4。著名な RedHat Enterprise Linux クローンディストリビューションの一つ。
NIS	ypserv 2.31, ypbind1.37, yp-tools 2.14。クラスタ内部のユーザ認証に使用。node0 が NIS サーバ。
NFS	v4。 node0 の/home と/common を node1～node6 がマウントするために使用する。
GCC	ver.4.8.5。GNU のコンパイラコレクション。C、Object C、C++、Fortran をインストール。
GMP	ver.6.0.0。著名な多倍長精度計算用ライブラリ。
MPFR	ver.3.1.1。著名な GMP ベースの多倍長精度浮動小数点演算ライブラリ。
ZKCM [9)]	執筆時点で ver.0.4.3 を使用しているが最新ベータ版に随時更新。著者が開発している多倍長精度の行列演算用 C++ライブラリ。なお量子計算シミュレーションライブラリの ZKCM_QC も導入。
CUDA	ver.9.1。NVIDIA 社が自社の GPU 向けに提供している GPGPU 用ライブラリ。
MPICH	ver.3.2。並列計算用の通信方式である MPI の標準的な実装の一つ。

の一部をリストで示す。その他、CentOS 7.4 に含まれるカーネルやデバイスドライバ、開発用の標準的なライブラリももちろん使用している。

以上でクラスタマシンの構成は示したが、次節では実際の構築過程を順々に述べていく。

3．構築の実際

構築作業のおおまかな手順としては、ネットワーク機材の設定をした後、管理ノードと計算ノードを兼ねる node0 を構築し、続いて計算ノード node1 を構築する。node0 と node1 だけでクラスタマシンとして正しく動作するようにソフトウェア環境を構築したら、node1 の HDD を複製機能付き HDD スタンドでコピーし、node2～node6 の HDD を作成する。この手順であれば、node2～node6 の構築作業は実質的に BIOS 設定

と部品のネジ止めおよび配線だけになり、効率的である。

（１） ネットワーク機材の設定

最初に VPN ルータの設定を行う。WAN 側ポートは学科内プライベートネットワークにつなぐ。このポートの IP アドレスは、学科内 DHCP サーバからの自動取得とした。ノード node0～node6 のソフトウェアインストール／アップデートのトラフィックのみがこのポートを流れるため、この設定で問題ない。LAN 側ポートには 192.168.201.1 の（クラス C の）プライベート IP アドレスを割り当て、node0～node6 のゲートウェイとする。また、node0～node6 の 10GBase-T NIC の MAC アドレスをあらかじめ控えておき（通常、ボード上のシールに記載がある）、DHCP の固定 IP アドレス割り当て設定でそれぞれの MAC アドレスに 192.168.201.100～192.168.201.106 を割り当てる。これで各ノードは起動時に DHCP で内部側の IP アドレスを取得することができる。

また、node0 の WAN 側 NIC の MAC アドレスも控えておき、ネットワーク管理者（本学では総合情報センター）に依頼して、グローバル IP アドレスの割り当て（202.16.yyy.yyy とする）と必要であれば基幹スイッチのルーティング設定をしてもらう。ネットワーク管理者から IP アドレスの通知を受けたら、node0 の WAN 側 IP アドレスは DHCP を使わずに固定の設定とする（次の小節を参照のこと）。

（２）node0 の構築

管理ノード兼計算ノードである node0 の構築について述べていく。ハードウェア構成は表-1 で示したとおりである。UPS は外付けで USB 接続している。

BIOS 設定 まず BIOS を以下のように設定した：(i) AC 電源断から復帰時にパワーオンとなるように設定した。これは、停電時の自動復帰のためである。(ii) Windows10 用の WHQL サポートを disable に設定した。また、Secure Boot 機能を disable、ブートのレガシーモードを enable、Fast Boot 機能を disable に設定した。これらは、CentOS 7.4 の起動の安定のためと、NVIDIA 社のグラフィックドライバーの動作のために必要な設定である。(iii) ブートデバイスの順番を、HDD が一番になるように設定した。これは、停電からの自動復帰時に DVD メディアや USB メモリからの起動を避けるためである。

OS のインストール 続いて、OS のインストール作業に入った。CentOS 7.4 は DVD のインストール用メディアを用意し、HDD２台を Linux カーネルのソフトウェア RAID 機能でミラーリングする設定でインストールした。/boot に 1GB、/home に 4,656GB、/ に 812GB の（HDD パーティションの組からなる）RAID 1 の md デバイスをマウントして使用する。ファイルシステムは XFS を採用した。また、SWAP 領域は（RAID にせずに）それぞれの HDD で 64GB を用意した。初期インストールソフトウェア群は、GCC や GMP、MPFR、Autoconf や Automake、GIT といった開発用のものと基本的なカーネルやネットワーク関連のユーティリティである。

SELinux の無効化 OS が HDD から起動するようになったら、まずは SELinux を無効にしておく。これは、クラスタマシンのユーザは常識的な利用を逸脱しないと想定されることと、ノード間やリモートホストからのファイル参照が旧来のファイルパーミッションのみの権限管理で直感的に行えるようにするためである。管理者権限で設定ファイル/etc/selinux/config を vi 等の適当なエディタで開く。

```
$sudo vi /etc/selinux/config
```

（以降の設定ファイル編集も同様に行う。）ファイル中に SELINUX=で始まる行があるので、

```
SELINUX=disabled
```

に変更して保存する。

デバイスドライバ類のインストール Intel 1GbE オンボード NIC はこの時点で動作するので、これを学内 LAN につなげて、デバイスドライバ類をインストールしていく。そのために Dynamic Kernel Module Support（DKMS）を有効にする必要がある。DKMS には、登録した

DKMS 対応ドライバモジュールは、カーネルアップデート時に自動再構築してくれる仕組みがある。この機能を使わなければ、アップデートする度にドライバを手動で入れ直すことになるので、運用上必須の機能である。CentOS で DKMS を使うためには、よく知られた追加パッケージレポジトリである ELRepo を利用する。本学ではプロキシ経由で外部との接続をしなければならないため、/etc/yum.conf に以下のような行を追記する：

```
proxy=http://[プロキシサーバ]:[接続ポート]
```

ただし [･･･] 部分は読み替えのこと。これで、パッケージ管理ソフト yum が外部アクセスできるようになった。実は、ELRepo 内のパッケージはしばしば EPEL レポジトリに依存しているので、こちらを先に利用できるようにしておいた方が良い。EPEL のウェブサイト https://fedoraproject.org/wiki/EPEL の指示にしたがって epel-release-latest-7 パッケージをインストールする。続いて、ELRepo のウェブサイト http://elrepo.org の指示にしたがって彼らの GPG キーと elrepo-release パッケージをインストールする。そして、コンソールから

```
$sudo yum install kernel-devel dkms
```

として dkms パッケージを入れた。

ではデバイスドライバのインストールに入る。NIC の AQR105 10GBase-T カードを動作させるため、A. Cooks 氏が公開している tn40xx ドライバモジュールを以下のようにして組み込む。

```
$sudo git clone -b release/tn40xx-001 ¥
  https://github.com/acooks/tn40xx-driver.git ¥
  /usr/src/tn40xx-001
$sudo dkms add -m tn40xx -v 001
$sudo dkms install -m tn40xx -v 001
```

なお他の 10GBase-T NIC の場合も標準ではドライバモジュールが組み込まれていないことが多く、ドライバのレポジトリを探して同様の手順でインストールすることになる。

続いて NVIDIA 社のプロプライエタリなグラフィックドライバをインストールする。NVIDIA 社のサイト https://www.geforce.com/drivers から Linux-64bit 用のドライバの run ファイルをダウンロードして保存しておく。この時点では ver.390.87 を使った。これを利用するには、すでにインストールされているグラフィックドライバである nouveau を無効にしておく必要がある。これには、/usr/lib/modprobe.d/nvidia.conf というテキストファイルを作り以下の一行を書く：

```
blacklist nouveau
```

また、テキストファイル/etc/default/grub の GRUB_CMDLINE_LINUX=で始まる行の右辺の最初のダブルクォーテーションの直後に、

```
nouveau.modeset=0 rd.driver.blacklist=nouveau
```

を追記する。これで PC を再起動すれば nouveau が無効の状態で立ち上がる。そうなっているかの確認は、

```
$lsmod | grep nouveau
```

で表示されなければ無効になっている。もし無効になっていない場合は、カーネルの initramfs イメージを以下のコマンドで再構成する必要がある。

```
$sudo dracut --force
```

再構成したら再起動しておく。では、nouveau が無効にできたとする。PC 起動時にシングルユーザモードにする（カーネルオプション single を指定）か、あるいは、

```
$sudo init 1
```

でシングルユーザモードに移行する。そうしてお

いて、ダウンロードしておいた run ファイルを実行し、実行後に表示される指示にしたがってインストールする。

```
#./NVIDIA-Linux-x86_64-390.87.run
```

ここで、DKMS モジュールを生成する選択肢を選んでインストールする。PC を再起動すれば、NVIDIA 社のドライバが動作しているはずである。確認するには、

```
$lsmod | grep nvidia
```

と打ち込んで、nvidia というモジュール名が表示されれば良い。

デバイスドライバ類としては最後に、UPS のドライバをインストールする。UPS はメーカが Linux ドライバ、ユーティリティを用意していることが多い。今回はオムロンソーシアルソリューションズ社が自社製 UPS の制御用に配布している PowerAttendant Lite ver.1.0 を使用する。https://www.oss.omron.co.jp/ups/から Linux 版を取得し、取扱説明書にしたがってインストールした。この制御ソフトには GUI があり、これを使って電源断からシャットダウン開始までの待機時間を 660 秒に設定した。node0 は管理ノードを兼ねていて電源断時には最後にシャットダウンする必要があるので、他のノードよりも 60 秒長い待機時間を設定している。

時刻合わせの設定　ここで、時刻合わせの設定を入れておく。まず ntpdate をインストールする：

```
$sudo yum install ntpdate
```

毎日時刻合わせをするように、以下の内容でテキストファイル/etc/cron.daily/ntptimesync を作成する。

```
#!/bin/bash
ntpdate ntp.cc.sojo-u.ac.jp
```

作成したら実行可能にしておく。

```
$sudo chmod a+x /etc/cron.daily/ntptimesync
```

また、起動時にも時刻合わせをするようにする。それには/etc/rc.d/rc.local に以下を追記する。

```
ntpdate ntp.cc.sojo-u.ac.jp
```

また、CentOS は ver.7 から rc.local がデフォルトでは実行可能でないパーミッションになっているため、実行可能に変更しておく：

```
$sudo chmod a+x /etc/rc.d/rc.local
```

hosts ファイルの記述　ではここで、以降の設定で他ノードを参照しやすくするために/etc/hosts ファイルに以下を追記しておく。

```
192.168.201.100 node0 node0.cluster.sncq.cis.sojo-u.ac.jp
192.168.201.101 node1 node1.cluster.sncq.cis.sojo-u.ac.jp
192.168.201.102 node2 node2.cluster.sncq.cis.sojo-u.ac.jp
192.168.201.103 node3 node3.cluster.sncq.cis.sojo-u.ac.jp
192.168.201.104 node4 node4.cluster.sncq.cis.sojo-u.ac.jp
192.168.201.105 node5 node5.cluster.sncq.cis.sojo-u.ac.jp
192.168.201.106 node6 node6.cluster.sncq.cis.sojo-u.ac.jp
```

これで node0 や node1 という短い名前でホストを参照できる。

NFS サーバの設定　引き続いて、node0 の一部ディレクトリを他のノードが NFS でマウントして使用できるようにする（後述のファイアウォールの設定も必要）。カーネルの NFS サーバを使うが、ユーティリティ類はインストールしなければならない：

```
$sudo yum install rpcbind libnfsidmap
$sudo yum install nfs-utils nfs4-acl-tools
```

なお、rpcbind は古いタイプの NFS 通信が RPC を使用するためインストールしている。次に NFS サーバの設定として、/etc/exports に以下を記述する。

```
/home node?(rw,sync) node??(rw,sync)
```

```
/common node? node??
```

ここでホスト名中クエスチョンマークは任意の文字 1 文字にマッチするワイルドカードである。そして以下の一連のコマンドで rpcbind と NFS サーバが起動時に立ち上がるようにするとともに、起動する。

```
$sudo systemctl enable rpcbind.service
$sudo systemctl start rpcbind.service
$sudo systemctl enable nfs.service
$sudo systemctl start nfs.service
$sudo systemctl enable nfs-server.service
$sudo systemctl start nfs-server.service
```

NFS を使って/home をノード間で共有するのはクラスタマシンでは典型的な NFS の使用法である。その他、上の/etc/exports には/common についての記述も書いた。これは、yum で管理できないソフトウェアはノードごとにインストールするとアップデートに手間がかかるので、node0 の/common に置いておき、NFS でマウントして利用することにしたいからである。なお/common については rw オプションなしで記述しているのでリードオンリーでの NFS マウントになり、計算ノードからは書き込みできない。

/common 以下にインストールしたライブラリの利用のため、環境変数 PATH に/common/bin を追記しておく。また、環境変数 LIBRARY_PATH と LD_LIBRARY_PATH ともに/common/lib と/common/lib64 を追記しておく。さらに、環境変数 C_INCLUDE_PATH と CPLUS_INCLUDE_PATH とも に /common/include を追記しておく。これには、/etc/profile.d/common.sh を作成して以下のように記述すれば良い。

```
export PATH=/common/bin:$PATH
export LIBRARY_PATH=/common/lib64:[改行空白なしでつづく]/common/lib:$LIBRARY_PATH
export LD_LIBRARY_PATH=/common/lib64:[改行空白なしでつづく]/common/lib:$LD_LIBRARY_PATH
export C_INCLUDE_PATH=/common/include:[改行空白なしでつづく]$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=/common/include:[改行空白なしでつづく]$CPLUS_INCLUDE_PATH
```

NIS サーバの設定　引き続き、今度は node0 で NIS サーバを動かして、他ノードのログオン時の認証を引き受けるように設定する。まずはパッケージのインストールをする：

```
$sudo yum install ypserv
```

次に/etc/sysconfig/network に以下のように追記する。

```
NISDOMAIN=cluster.sncq.cis.sojo-u.ac.jp
HOSTNAME=node0.cluster.sncq.cis.sojo-u.ac.jp
```

ここで NIS サーバを PC 起動時に立ち上がるようにし、また、起動しておく：

```
$sudo systemctl enable ypserv.service
$sudo systemctl start ypserv.service
```

続いて NIS サーバの初期設定を行う。次のコマンドを打ち込む。

```
$sudo /usr/lib64/yp/ypinit -m
```

そうすると、node0 の他に NIS サーバを走らせるホストがあるか聞いてくるが、今回の構成では node0 のみであるので、"next host to add:"というプロンプトに<Ctrl-D>を入力する。NIS サーバのリストとして node0 だけで良いか聞いてくるので、y を入力する。これで初期設定が終わったので、念のため NIS サーバを再起動しておく。

```
$sudo systemctl restart ypserv.service
```

さらに、ユーザが yppasswd コマンドでパスワード変更できるように yppasswdd を PC 起動時に立ち上がるようにし、また、起動しておく。

```
$sudo systemctl enable yppasswdd.service
$sudo systemctl start yppasswdd.service
```

また、node0 はクライアントを兼ねるので、NIS クライアントとしての設定も忘れずに入れておく。これは次を参照。なお、NIS の制御下ではユーザ作成はわずかに手間がかかるが、これについては小節（５）で述べる。

NIS クライアントの設定　NIS クライアントの動作に必要なパッケージをインストールする：

```
$sudo yum install ypbind yp-tools
```

設定ファイルは二つあり、まず/etc/yp.conf に次のように記述する。

```
domain cluster.sncq.cis.sojo-u.ac.jp server [改行なしでつづく] node0.cluster.sncq.cis.sojo-u.ac.jp
```

この設定の書式は、NIS ドメインについて、
domain　ドメイン　server　サーバホスト名
となっている。次に/etc/nsswitch.conf を編集する。このファイルの各行のうち、passwd、shadow、group、hosts で始まる行を次のように変更する。

```
passwd:     nis files sss
shadow:     nis files sss
group:      nis files sss
hosts:      files dns myhostname
```

それから ypbind サービスを起動時に立ち上がるようにし、また、起動しておく：

```
$sudo systemctl enable ypbind.service
$sudo systemctl start ypbind.service
```

これで、node0 は（後で設定する他のノード同様）ユーザ認証に NIS を優先的に使う。ユーザは node0 にログオン中に yppasswd コマンドで NIS のパスワードを変更できる。

ネットワークとファイアウォールの設定　次はネットワーク接続とファイアウォールの設定に入る。クラスタマシン内部の通信はすべて信頼できるとして trusted ゾーンに入るようにする。まず、10GBase-T NIC の接続名を分かりやすいものに変更する。次のコマンドを打ってネットワーク設定用のユーザインタフェースを出す：

```
$sudo nmtui
```

接続名（プロファイル名）は"有線接続１"となっていたが、"10GbpsLocal"に変更した。また、アドレスは 192.168.201.100/24 を手動設定し、ゲートウェイと DNS サーバは VPN ルータの LAN 側アドレス 192.168.201.1 を設定した。また、ルーティング設定は、デフォルトルートに使用しない、自動的に取得されたルートと DNS パラメータを無視する、という設定とした。ここで、手動設定にしたのは、node0 では NIC２枚とも使い、ルーティングが自動では適切に設定されないためである。ついでに、オンボード 1GbE NIC の設定もした。こちらはアドレスをネットワーク管理者に割り当ててもらった 202.16.yyy.yyy（一部伏せている）とし、デフォルトルートに設定した。では次に、firewalld を PC 起動時に立ち上がるようにし、また、起動しておく：

```
$sudo systemctl enable firewalld.service
$sudo systemctl start firewalld.service
```

続いてファイアウォールの設定を以下のように入れる：

```
$sudo firewall-cmd --zone=trusted [改行なしでつづく]--change-interface=10GbpsLocal --permanent
$sudo firewall-cmd --zone=trusted [改行なしでつづく]--add-source=192.168.201.0/24 --permanent
$sudo firewall-cmd --reload
```

なお設定が正しく入っているかは次のコマンドで確認できる：

```
$sudo firewall-cmd --zone=trusted --list-all
```

以下のような表示が出てくれば正しく設定できている。

```
trusted (active)
  target: ACCEPT
  icmp-block-inversion: no
  interfaces: 10GbpsLocal
  sources: 192.168.201.0/24
        [以下省略]
```

trusted ゾーンに 10GbpsLocal が入っていることが分かる。なお、もう一枚の NIC はデフォルトのまま public ゾーンに入っている。

SSH サーバの設定　続いて、SSH サーバを立ち上げてリモート接続可能にしておくとともに、アクセス制限もかける。まず SSH サーバが PC 起動時に立ち上がるようにし、また、起動する。

```
$sudo systemctl enable sshd.service
$sudo systemctl start sshd.service
```

SSH サーバの設定ファイル/etc/ssh/sshd_config に以下の二つの設定を追加する。

```
PermitRootLogin no
PubkeyAuthentication yes
```

それぞれ、ルートログインの拒否と、公開鍵認証でのログインの許可を意味する。続いて /etc/hosts.deny と/etc/hosts.allow の二つのファイルを使って、NFS アクセスと SSH アクセスの制限をかける。NFS はクラスタマシン内、SSH は学内アクセスのみ許容する設定とした。

/etc/hosts.deny の追記部分は以下：

```
nfsd: all
rpcbind: all
mountd: all
sshd: all
```

/etc/hosts.allow の追記部分は以下：

```
nfsd: localhost
nfsd: 202.16.yyy.yyy
nfsd: 192.168.201.
rpcbind: localhost
rpcbind: 202.16.yyy.yyy
rpcbind: 192.168.201.
mountd: localhost
mountd: 202.16.yyy.yyy
mountd: 192.168.201.
sshd: 192.168.
sshd: 172.16.0.0/255.240.0.0
sshd: 202.16.zzz.0/255.255.240.0
```

ここで、202.16.yyy.yyy は前述したように管理者に割り当ててもらったアドレスであり、一部伏せてある。また、sshd について許容するアクセス元として、クラス C、クラス B のプライベートアドレスの範囲の他、本学のグローバルアドレスの範囲（やはり一部伏せてある）も指定している。

標準的な開発ツールのインストール　では、科学技術計算用に使うプログラムライブラリ類のインストールに移る。標準的な開発用のライブラリとソフトウェアで、簡単にインストールできるものを最初に入れておく。

```
$sudo yum install gcc-gfortran gcc-c++
$sudo yum install gcc-objc gcc-objc++
$sudo yum install gdb
$sudo yum install gmp* mpfr* lapack*
$sudo yum install glibc-static
$sudo yum install blas* atlas* boost*
$sudo yum install fftw fftw-devel fftw-static
$sudo yum install gnuplot
$sudo yum install valgrind valgrind-devel
$sudo yum install doxygen
```

MPICH のインストール　これで標準的なソフトウェアが入ったので、まずは MPICH をインストールする。

```
$sudo yum install mpich-3.2 mpich-3.2-doc [改行なし
```

```
でつづく］ mpich-3.2-devel　mpich-3.2-autoload
```

ユーザが MPICH を使いやすいように、クラスタを構成するマシンのリストを書き込んだ簡便なマシンファイルを用意しておく。テキストファイル/common/conf/mpi_machinefile を作成し、以下を記述する。

```
node0
node1
node2
node3
node4
node5
node6
```

また、ユーザが MPICH を使った実行ファイルの実行時にこのマシンファイルを探さなくてよいように、あらかじめ alias を設定する。ファイル/etc/profile.d/mpi_machinefile.sh を以下の内容で作成する。

```
alias mpirun='mpirun -machinefile ［改行なしでつづく］　/common/conf/mpi_machinefile'
```

続いて、クラスタ内部の SSH アクセスは事前のホスト fingerprint の認知をしなくても良い設定にしておく。この設定をしないと、MPI プログラムを走らせる前にユーザがノード間の fingerprint の認知をさせねばならなくなる。設定としては、/etc/ssh/ssh_config に以下を追記する。

```
Host 192.168.201.* node*
   CheckHostIP no
   StrictHostKeyChecking no
   LogLevel=quiet
   UserKnownHostsFile=/dev/null
```

なお、この設定が正しく入っていなければ、ユーザが MPI プログラム実行時に Host key verification failed というエラーが発生する。

CUDA のインストール　次に、GPGPU 用に NVIDIA 社が提供している CUDA ライブラリをインストールする。CUDA のダウンロードサイト https://developer.nvidia.com/cuda-toolkit-archive から、グラフィックドライバのバージョンに合わせたバージョンの CUDA をダウンロードする。今回は、ver.9.1 の CUDA を選び、Linux x64 CentOS 7用の network インストールファイル cuda-repo-rhel7-9.1.85-1.x86_64.rpm を選択した。以下のコマンドでインストールする。

```
$sudo yum install libvdpau
$sudo rpm -Uvh cuda-repo-rhel7-9.1.85-1.x86_64.rpm
$sudo yum install cuda
```

また、/etc/profile.d/nvidia-cuda.sh を作成して以下の内容を記述してパスを通しておく。

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:
［改行空白なしでつづく］$LD_LIBRARY_PATH
```

ここで、LD_LIBRARY_PATH の記述があるのに LIBRARY_PATH は記述していないのは、cuda のライブラリファイルの中には名前が cu で始まらないものがあって、リンク時に自動でこれらが見つかると他のライブラリと競合する可能性があるためである。ただ、その可能性は低いと考える管理者は、LIBRARY_PATH についても記述して良いだろう。

なお、NVIDIA 社の開発者向け議論サイト（https://devtalk.nvidia.com/）では使用する GPU によっては nvidia-settings という GUI ツールでのドライバ設定やドライババージョンによる CUDA の計算の安定性の違いがしばしば議論されるが、今回は特段問題なかった。

以上で、node0 の環境構築が完了したので、念のため再起動して正しく立ち上がることを確認しておく。

（3）node1 の構築

計算ノードの構築に入る。まずは node1 を構築し、動作確認してから残りのノードはそのコピー

として構築する。node1 の基本的な構築方法は前の小節で述べた node0 の場合と同様であるが、一部、サーバではなくクライアントとして動作させるため設定が異なる部分がある。以下、手順を追って説明する。

BIOS 設定　node0 と同じである。

OS のインストール　node0 と同じ手順であるが、マウントポイントと HDD パーティションの容量は/boot に 1GB、/ に 913GB、また SWAP 領域に 64GB とした。なお後述のように、/home は NFS で node0 からマウントして使うようにするが、最初の設定時点では当然ローカルディスク上の/home が見えていて、ローカルユーザで PC にログインして設定していく。

SELinux の無効化　node0 と同じである。

デバイスドライバ類のインストール　node0 と同じであるが、最後の UPS 関連の設定だけが異なり、電源断からシャットダウン開始までの待機時間を 600 秒に設定した。これは、停電時に管理ノードを兼ねる node0 よりも 60 秒先に停止することで NFS サーバへの通信エラーを回避するためである。

起動時の遅延設定　NFS でマウントするマウントポイントがある関係で起動時に管理ノードよりも遅れて立ち上がる必要がある。そのために、/etc/grub2.cfg を編集して、"set timeout=5"となっている箇所をすべて"set timeout=50"に書き直す。

時刻合わせの設定　node0 と同じである。

hosts ファイルの記述　node0 と同じである。

NFS クライアントの設定　node1 では/home と/common のマウントポイントについては、NFS で node0 上のディレクトリをマウントして使用する。そのために、NFS クライアントの設定を行う。設定に先立ってやはり NFS 関連のユーティリティ類をインストールしておく：

```
$sudo yum install rpcbind libnfsidmap
$sudo yum install nfs-utils nfs4-acl-tools
```

設定としては、/etc/fstab に以下の２行を追記する。

```
node0:/home /home nfs rw 0 0
node0:/common /common nfs rw 0 0
```

また、rpcbind を PC 起動時に立ち上がるようにし、起動もしておく。

```
$sudo systemctl enable rpcbind.service
$sudo systemctl start rpcbind.service
```

これで、後述するネットワークとファイアウォールの設定を入れてから再起動すれば、node0 の/home と/common をマウントして使うようになるが、その前に NIS クライアントの設定に移る。

NIS クライアントの設定　node1 のログイン時の認証を node0 の NIS サーバに依頼するようにする。設定は、node0 の NIS クライアント設定で述べたものとまったく同じである。

ネットワークとファイアウォールの設定　引き続いて node1 のネットワーク設定を行う。ネットワーク設定用のユーザインタフェースを出す:

```
$sudo nmtui
```

10GBase-T の NIC の接続名（プロファイル名）を"有線接続 1 "から"10GbpsLocal"に変更する。ただし、node0 のときとは異なり、IP アドレスは DHCP サーバからの自動取得のままとした。これは、node1 の HDD をコピーするだけで計算ノードを増やせるようにするためである。計算ノードのデフォルトゲートウェイは VPN ルータであるため、デフォルトルートもそのまま自動設定とした。なおここで、10GBase-T NIC のデバイス名を控えておく（enp3s0 となっていたが、もちろん環境に依る）。また、オンボード 1GbE NIC については計算ノードではケーブルを接続しないので設定は特段入れなかった。

ファイアウォールの設定は node0 のときと同様であり、まず

```
$sudo systemctl enable firewalld.service
$sudo systemctl start firewalld.service
```

としてサービスを有効化してからファイアウォールの設定を入れる。node0 のときに比べて 1 行多い。

```
$sudo firewall-cmd --zone=trusted [改行なしでつづく]--change-interface=10GbpsLocal --permanent
$sudo firewall-cmd --zone=trusted [改行なしでつづく]--add-source=192.168.201.0/24 –permanent
$sudo firewall-cmd --zone=trusted [改行なしでつづく]--change-interface=enp3s0 --permanent
$sudo firewall-cmd --reload
```

接続 10GbpsLocal だけでなく、そのデバイス enp3s0（適宜読替のこと）を追加で trusted ゾーンに加えるのは冗長であるのだが、今回の構築では node1 ではこの接続は NetworkManager (nm) の自動設定の影響下にあるので、デバイス名を指定してゾーンに加えておいた。

SSH サーバの設定　node0 と同じ手順で SSH サーバを有効化して起動しておく。また、/etc/ssh/sshd_config も同様に node0 と同じ変更を加え、公開鍵によるログオン認証を許可しておく。次に、/etc/hosts.deny と/etc/hosts.allow への追記によるアクセス制限は次のようにする。

/etc/hosts.deny への追記：

```
sshd: all
```

/etc/hosts.allow への追記：

```
sshd: 192.168.201.
```

これらの設定で、計算ノードへの SSH アクセスをクラスタ内部に限定している。

標準的な開発ツールのインストール　node0 と同じである。

MPICH のインストール　node0 と同じである。説明の後の方で述べた ssh_config の設定までしっかり設定を入れておく。

CUDA のインストール　node0 と同じである。

（４）node2～node6 の構築

node1 の構築が終わった段階で、node0 と node1 の２台だけでクラスタマシンとして動作するか確かめておく。node0 と node1 を起動してみて、後述する（５）の手順でテストユーザを作ってから、node1 でそのユーザでログオンしてみる。NFS と NIS が正しく動作しているか、以下のように確認する。まず、

```
$ypwhich
```

と打ち込んで node0 と表示されれば NIS が正しく動いている。続いて、

```
$mount
```

と打ち込んでみて、

```
[上側省略]
node0:/common on /common type nfs4 ([右側省略]
node0:/home on /home type nfs4 ([右側省略]
[下側省略]
```

と表示されれば NFS が正しく動作している。

一度両ノードをシャットダウンし、node1 の複製作業に入る。node1 の HDD を HDD 複製機（CFD 販売の KURO-DACHI/CLONE/U3 を使用した）で複製して node2～node6 の HDD を作成した。複製には 1 台あたりおおよそ 200 分を要した。PC5 台にこれら HDD をそれぞれ搭載し、BIOS を前述の node0 のときと同じ BIOS 設定にした。HDD が node1 の複製なので OS 設定以降の作業は不要であり、node2～node6 として正常に動作した。

（５）ユーザの作成

NIS によるログイン認証を採用しているため、ユーザ作成にはわずかに手間がかかり、以下の手順となる（ユーザ名を仮に testuser としている）。

```
$sudo useradd testuser
$sudo passwd testuser
  [プロンプトで初期パスワードを設定する]
$cd /var/yp
```

```
$sudo make
```

また､ユーザがログインパスワードを変更するには、通常の passwd コマンドではなく、

```
$yppasswd
```

を使用する｡これはユーザに伝えておくべきである。

ユーザ作成に関連して､ユーザが最初にログインしたときを考えると、ユーザはまず MPI プログラムの実行のための初期設定を行うことになる｡すなわち､公開鍵認証で全ノードにパスフレーズ入力なしで SSH ログインできるように設定しなければならない。NFS で/home をマウントしているのでそれほど手間ではないのだが､ユーザの利便性を考えて､これが簡単に行えるスクリプトを用意しておくべきである。ファイル /common/bin/cluster_setup_auth_keys.sh を作成し、以下の内容で保存する。

```
#!/bin/bash
mkdir -p $HOME/.ssh
chmod 700 $HOME/.ssh
cd $HOME/.ssh
ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N " [注]
cat id_rsa.pub >> authorized_keys
chmod 600 authorized_keys
echo "Your public key has been added to [改行なしでつづく] $HOME/.ssh/authorized_keys."
echo "Setup of $HOME/.ssh/authorized_keys [改行なしでつづく] has been done."
```

ここで[注]は5行目の最後についての注意であり、-N <空白><シングルクォーテーション><シングルクォーテーション>である。では、このスクリプトを実行可能にしておく：

```
$sudo chmod a+x /common/bin/cluster_setup_auth_keys.sh
```

ユーザには、最初のログイン時に cluster_setup_auth_keys.sh を実行するように伝えておくと良い。あるいは、管理者があらかじめ代理でログインしてこのスクリプトを実行しておくのが親切かもしれない。

（6）簡単な動作テスト

上述の手順でテスト用のユーザ（ここでも testuser とする）を作成したら、簡単な動作テストを行う。学内の他ホストから SSH で node0 のグローバル IP アドレスへアクセスしてログインする。ログインしたら

```
$cluster_setup_auth_keys.sh
```

を実行する。これで MPI の実行環境は整ったはずである。以下のサンプルコード test0.c を作成する。

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[ ])
{
  int s, r;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &s);
  MPI_Comm_rank(MPI_COMM_WORLD, &r);
  printf("Process %d / %d processes¥n", r, s);
  MPI_Finalize();
  return 0;
}
```

ここで､関数名の大文字小文字は打ち間違いではなく、このとおりである。では、以下のようにコンパイル～実行をする。

```
$mpicc -o test0 test0.c
$mpirun ./test0
```

MPI 環境が正しく構築されていれば、以下のような出力が出る。

```
Process 5 / 7 processes
Process 3 / 7 processes
```

```
Process 1 / 7 processes
Process 0 / 7 processes
Process 4 / 7 processes
Process 2 / 7 processes
Process 6 / 7 processes
```

表示されるプロセスの順番は実行毎に異なる。なお、本稿では mpirun は前述のようにマシンファイル/common/conf/mpi_machinefile を使うように alias してある。別のマシンファイルを使いたいユーザは、一度

```
$unalias mpirun
```

で unalias しておき、ユーザ作成のマシンファイル（仮に hogemf とする）を指定して実行する：

```
$mpirun -machinefile hogemf ./test0
```

続いて、GPGPU 環境のテストについてごく簡単に説明する。NVIDIA 社のサンプルコードを指定ディレクトリに展開するコマンドが CUDA には標準で用意されている。

```
$cuda-install-samples-9.1.sh .
```

とすると、カレントディレクトリ（．）に NVIDIA_CUDA-9.1_Samples というディレクトリが作成され、その中にサンプルコードが入ったサブディレクトリが分類されて並ぶ。手始めに、GPU カード情報を表示するサンプルを実行してみる：

```
$cd NVIDIA_CUDA-9.1_Samples/ 1_Utilities/deviceQuery
$make
$./deviceQuery
```

CUDA 環境が正しく構築されていれば、"Detected 1 CUDA Capable device(s) "で始まる GPU 検出情報が表示される。

以上で、MPI と CUDA の簡単な動作テストができた。

（7）追加ソフトウェアのインストール

管理者が提供していないソフトウェアは、各ユーザがホームディレクトリにインストールして使うのが通常である。/home は NFS で全ノードがマウントしているので、特段問題は発生しないであろう。しかし、多くのユーザが同じソフトウェアを必要とする場合は、運用開始後でも管理者が追加で提供するのが親切である。もしそのソフトウェアが rpm パッケージで提供されているのであれば、各ノードで同じコマンド（$sudo yum install <パッケージ>）を実行すればよい。ノード数の分実行しても大した手間ではない。

問題はソースコードのパッケージで提供されている場合で、各ノードごとにコンパイル～インストールしていては手間がかかりすぎる。この場合は、/common 以下にインストールすればよい。/common は NFS で全ノードがマウントしており、パスも通しているので、ここを起点にする（前述の node0 の NFS サーバの設定を参照のこと）。

ソースコードのコンパイル～インストールについては、大半の計算科学系ソフトウェアは GNU の Autoconf を使用しており、./configure スクリプトでインストールの前処理をする典型的なスタイルがある。一例として、私が開発している ZKCM [9]ライブラリの ver.0.4.3 の tar.gz パッケージ（ダウンロード元 URL：http://zkcm.sf.net）は次のようにインストールできる。

```
$tar xfz zkcm_lib-0.4.3.tar.gz
$cd zkcm_lib-0.4.3
$./configure --prefix=/common
$make
$sudo make install
$cd zkcm_cus
$./configure --prefix=/common ¥
  --with-zkcm-include=/common/include
$make
$sudo make install
```

このように、たいてい、./configure スクリプトのオプションで--prefix=/common と指定するこ

とで/common 以下にインストールできる。また、--with-hogehoge-include=/common/include のようなオプション指定は、hogehoge パッケージが既にインストールされていてそのインクルードファイルが/common/include にあるということを言っている。このような--with で始まるオプションについては、./configure --help でヘルプを表示するとそこに書かれている場合が多い。なお上の例で zkcm_cus というのは、パッケージ内パッケージで、CUDA を一部利用する追加関数を ZKCM に導入するためのサブライブラリである。また今回、ZKCM の関連ライブラリである量子コンピュータのシミュレーションライブラリ ZKCM_QC もインストールしたのであるが、これについては本稿の主題から逸脱するので割愛する。

（8）ネットワークの性能の確認

ノード間は 10Gbit イーサネットで結んでいるので、理論上は最大 10Gbits/sec の帯域があるはずであるが、念のため実際の帯域を測定しておく。今回は帯域測定ソフトとして著名な iPerf の バージョン 3.1.3 を iPerf のウェブサイト（https://iperf.fr）からダウンロードして使用した。まずは、node0 の適当なディレクトリでソースコードのアーカイブを展開して以下のようにコンパイル、インストールする。前小節で述べたように/common 以下にインストールしている。

```
$tar xfz iperf-3.1.3-source.tar.gz
$cd iperf-3.1.3
$./configure --prefix=/common
$make
$sudo make install
```

続いて、測定に入る。node0 で iPerf サーバをデーモンとして起動する。

```
$iperf3 -s -D -f g > iperf3log
```

ここで、オプション-s はサーバ、-D はデーモン、-f g は Gbit 単位を意味し、> で出力を適当なログファイルへリダイレクトしている。そうしておいて、node1 にログオンして、iPerf クライアントを走らせて帯域を測定する。

```
$ssh node1
$iperf3 -c node0 -f g
      [クライアント出力は省略]
$exit
```

ここで、２行目中のオプション-c はクライアントを意味し、その後の node0 は、node0 で起動している iPerf サーバへ通信することを意味する。クライアントの出力結果の中に帯域が“Bandwidth”として表示され、“sender”、“receiver”項目ともに 10 秒間の平均値“9.40 Gbits/sec”が得られた。node2～node6 での測定も同様であり、まれに 0.01Gbits/sec だけ上下することがあったのみである。計測された平均帯域 9.40Gbits/sec は機材の上限に近い十分な数値と言える。なお、計測後は node0 で iPerf サーバを終了しておく：

```
$pkill iperf3
```

（9）ノードを後から追加するには

本稿ではノードは node0～node6 の 7 台である前提で構築した。構築後にノード、例えば node7 を追加するには、まず VPN ルータの DHCP 設定に追加をし、続いて各種設定ファイルの中で node0～node6 までについて書かれているものに node7 の分を追記し、その後 node1 の HDD を複製して node7 用に使うことになる。より具体的には、

(i) VPN ルータの DHCP の固定 IP アドレス設定に node7 の分を追加する、

(ii) 全ノードの/etc/hosts ファイルに node7 の分を追記する、

(iii) node0 の/common/conf/mpi_machinefile ファイルに node7 の分を追記する、

(iv) node1 の HDD の複製を作って node7 に搭載する、

の手順となる。なお、10GBase-T スイッチングハブのポート数が足りなくなるが、それにはハブ

を追加するか、よりポート数の多いものに取り替えれば良い。

また、ノードが非常に増えて node99 を超える番号のノードを追加することになった場合には、node0 の/etc/exports ファイルに番号３文字分のワイルドカード付きの設定を追記してから NFS サーバを再起動する（$sudo systemctl restart <サービス名> でサービスを再起動できる）という手順が増える。ただ本稿のクラスタマシン構成ではそこまでノード数が増えることは想定していない。ノード数および／またはユーザ数が大きくなるとリソース全体を一部のユーザが占有可能なのは不公平となり、ジョブの管理が必要となるが、これについては５節で簡単に議論する。

以上、本節では小規模クラスタマシンの構築から動作テストまでの過程を詳細に述べた。小節（９）ではノードの追加方法も述べたが、執筆時点では７ノードのまま追加していない。次節では、このクラスタマシンの性能を標準的な測定方法で評価する。

４．性能評価

伝統的に High Performance Computing（HPC）分野では性能評価には、行列が密な連立一次方程式を解く速度を評価するベンチマークソフトウェアである LINPACK [10] が用いられ、近年は最近のアーキテクチャに対応した LINPACK の実装の一つである HPL [8] が使用されている [11]。ベンチマークテストの結果は（倍精度の）実数演算の秒あたりの実行回数（flops）で得られる。GPGPU に対応した実装では NVIDIA 社が HPL を自社の GPU 用に拡張したもの [12] を提供しており [13]、今回はユーザ登録すればダウンロードできるバージョンである CUDA Accelerated LINPACK 1.5（hpl-2.0_FERMI_v15）を使用した。なお、今回ベンチマークソフトウェアはクラスタマシンの一般ユーザには提供しないので、以下ではもっぱらテストユーザのホームディレクトリ以下でビルドして使用している。

（１）ベンチマークの準備

まずはベンチマークソフトウェアの依存関係のため GotoBLAS2 ライブラリをコンパイルする。すでに開発元の Texas Advanced Computing Center（TACC）は GotoBLAS2 の更新を停止していて OpenBLAS が後継ライブラリと目されているが、今回は GotoBLAS2 の Ver.1.13 を TACC のウェブサイト [14] からダウンロードして使用した。次のように適当な展開場所でパッケージ展開、場所移動する。

```
$cd [展開場所]
$tar xfz GotoBLAS2-1.13.tar.gz
$cd GotoBLAS2
```

ここでコンパイルに移るのであるが、最適化のために CPU アーキテクチャを指定する。本稿では使用 CPU が Intel NEHALEM 系統であるのでコンパイルターゲットとして NEHALEM を指定する。ただし、コンパイル時に単に TARGET=NEHALEM を指定して make すると、アセンブリファイル gemm_ncopy_4.S に関する invalid operands エラーが出る。また、オブジェクトリンク時にライブラリパスに"-l -l "が混入してストップするという不具合もある。そのため、以下のようにする。まず、getarch.c を編集し、コメントアウトを一つ外す：

```
49 行目 #define FORCE_NEHALEM
```

次に exports/Makefile を以下のように編集する：

```
先頭に挿入 FEXTRALIB_MOD = [改行なしでつづく] `echo $(FEXTRALIB) | sed 's/-l -l //'` [改行]
それ以降の部分 $(FEXTRALIB) をすべて $(FEXTRALIB_MOD)に書き換える。
```

そして、GotoBLAS2 ディレクトリ直下で以下のようにしてコンパイルする。

```
$make  http_proxy=http://[プロキシ]:[接続ポート]
```

プロキシを指定しているのは、ビルド途中で linpack-3.1.1 を外部からダウンロードしてくる

が、図-2 のとおり学内プロキシ経由になるためである。正しくコンパイルできれば、libgoto2_nehalemp-r1.13.so といった名前のファイルができているはずである。

続いて、前述の hpl-2.0_FERMI_v15 の tgz アーカイブファイルをウェブサイト[13] からダウンロードし、以下のように適当な展開場所で展開してから展開後のディレクトリへ移動する。

```
$cd [展開場所]
$tar xfz hpl-2.0_FERMI_v15.tgz
$cd hpl-2.0_FERMI_v15
```

そして CUDA_LINPACK_README.txt の指示にしたがってディレクトリ中の Make.CUDA を以下のように編集した。変更行のみ示す。

```
104 行目  TOPdir = [展開場所]/ hpl-2.0_FERMI_v15
119 行目  MPdir = /usr/lib64/mpich
120 行目  MPinc = -I/usr/include/mpich-x86_64
132 行目  LAdir = [GotoBLAS2 展開場所]/GotoBLAS2
133 行目  LAinc = -I$(LAdir)
136 行目  LAlib = -L$(TOPdir)/src/cuda -ldgemm [改行なしでつづく] -L/usr/local/cuda/lib64 -lcuda -lcudart [改行なしでつづく] -lcublas -L$(LAdir) -lgoto2 [改行なしでつづく] -lgfortran -lpthread
```

また、src/cuda/Makefile を以下のように編集した。やはり変更行のみ示す。

```
41 行目  DEFINES += -DGOTO
```

さらに、これは指示にないことであるが、src/cuda/cuda_dgemm.c には不備があり[15]、以下のような箇所が 4 箇所ある。

```
#ifdef GOTO
        [省略]
#endif
#ifdef ACML
        [省略]
#else
        [省略]
#endif
```

このようになっている箇所をすべて以下のように修正する（太字部分を追記）。

```
#ifdef GOTO
        [省略]
#endif
#ifndef GOTO
#ifdef ACML
        [省略]
#else
        [省略]
#endif
#endif
```

編集が終わったらパッケージをビルドする：

```
$cd [展開場所]/ hpl-2.0_FERMI_v15
$make
```

ビルドできたら、ベンチマークテストを実施するために、

```
$cd bin/CUDA
```

と打ち込んで場所を移動する。テスト実施時に共有ライブラリエラーを回避するため、そこにあるスクリプト run_linpack の中の環境変数 LD_LIBRARY_PATH に、[GotoBLAS2 展開場所]/GotoBLAS2 をコロン区切りで追記しておく。

（2）予備的なベンチマークテストによる調整

まずは単一ノードで小規模行列についての予備的なテストを走らせて GPU と CPU の使用比率を調整する。調整すべき値は、run_linpack スクリプトの中の環境変数 CUDA_DGEMM_SPLIT と CUDA_DTRSM_SPLIT であり、前者は dgemm、後者は dtrsm という名前の関数について GPU へ割り振る割合を決める。後者の値は（前者の値－0.10）とすることが推奨されている。スクリプト中の他の環境変数および HPL.dat フ

ファイル中のパラメータを表-3 のとおり固定して、（前者，後者（＝前者－0.10））の組を 0.01 刻みで動かしながらベンチマーク測定した。なお HPL.dat の “device out” 項目に “file”、“output file name” 項目に適当なファイル名を指定することで結果をファイルに保存した。実行コマンドとしては、最初に

```
$unalias mpirun
```

としてから、それぞれの値の組を設定した後

```
$mpirun -np 1 -hostfile hostfile ./run_linpack
```

を実行すればよい（ここで hostfile は実行ノードを各行 1 つ書いたテキストファイルであり、調整時は単に、node0 とだけ記述されたものである）。オプション -np 1 で MPI プロセス数を 1 にしているが、これは、hpl-2.0_FERMI_v15 では 1 GPU あたり 1 MPI プロセスを充てることになっているからである。なお実際には設定と実行を入れ込んだ計測用スクリプト（中身は自明な繰り返し構文であるので割愛する）を組んで実行した。（0.54，0.44）のときに計算所要時間は極小値 15.81 秒、演算性能は極大値 142.3Gflops をとった（表-4）。なお表-3 の GOTO_NUM_THREADS は 12 ではなく 6 として CPU のハイパースレッディングを使わない方が高速になる可能性もあったが、今回は 6 にした場合は、所要時間がほぼ同じかやや遅くなった（例えば（0.54，0.44）のときは 15.82 秒、142.2Gflops となった）。

表-3 調整時のその他パラメータの値

ファイル	パラメータ	値
run_linpack	CPU_CORES_PER_GPU	6
	GOTO_NUM_THREADS	12
HPL.dat	N	15,000
	NB	768
	P	1
	Q	1
	NBMIN	2
	PFACT	Left
	RFACT	Left
	BCAST	1ring
	L1	no-transposed
	U	no-transposed

表-4 調整時の測定結果

環境変数値の組*	所要時間[秒]	Gflops
(0.00，0.00)	28.33	79.42
(0.10，0.00)	18.11	124.3
	～中略～	
(0.51，0.41)	15.98	140.8
(0.52，0.42)	15.84	142.1
(0.53，0.43)	15.89	141.6
(0.54，0.44)	15.81	142.3
(0.55，0.45)	15.89	141.6
(0.56，0.46)	15.93	141.3
(0.57，0.47)	15.97	140.9
	～後略～	

傍注*（CUDA_DGEMM_SPLIT，CUDA_DTRSM_SPLIT）

以上によって、CUDA_DGEMM_SPLIT と CUDA_DTRSM_SPLIT の値の組は（0.54, 0.44）とすることに決めた。

（3）ベンチマークテストの実行と結果

調整が終わったので、続いて使用ノード数を 1 から 7 まで増やしながらベンチマークテストを実施する。それにあたっては以下の点を考慮する。

まず HPL では行列サイズ N が大きいほど処理能力が大きく計測される傾向にあるが、行列のデータとしての量 N×N×8 [バイト] がシステム全体のメモリ上で管理できる大きさでなければならない。今回の構成ではメインメモリが各ノード 32GB であり、単一ノードでの計測では N はおおよそ 65,000 が上限、7 ノードでの計測ではおおよそ 170,000 が上限となる。ただし常駐デーモンや通信サービス等が各ノード 1GB～2GB のメモリを消費するため、上限に近い場合スラッシングが発生する。そのため、単一ノード 60,000 程度、7 ノード 160,000 程度が N の妥当な設定値である。なお、演算性能の測定が目的の場合、N を固定する制約はないから、ノード数に応じて N を変化させるのが通常である [5]。

次に HPL ではプロセス数 NP は、NP=P×Q となるように設定する。P と Q は HPL.dat の中で設定し、NP はコマンドラインで実行時に以下のように与える（[NP]は 1 や 7 といった整数）。

```
$mpirun -np [NP] -hostfile hostfile ./run_linpack
```

hostfile には、使用するホスト名を列挙する。今回は例えば NP=7 のときは以下のようになる。

```
node0
～中略～
node6
```

構成上１ノードあたり１GPU なので以上のようになる。（もしそうでない場合は NP=P×Q が使用する GPU の数と一致するように設定・実行する。）

以上を考慮に入れて、プロセス数 NP=P×Q に対して所要時間と演算性能を測定した結果を表-5 にまとめる。使用した N の値は、[（NP の平方根）×60,000]を最も近い整数に丸めたものである。なお、環境変数 CUDA_DGEMM_SPLIT と CUDA_DTRSM_SPLIT の値の組は小節（２）で決めたとおり（0.54，0.44）とし、表-5 に示していない変数値は表-3 のままとした。

表-5　演算性能の測定結果

NP	P	Q	N	所要時間[秒]	Gflops
1	1	1	60,000	679.9	211.8
2	1	2	84,853	1,009	403.6
3	1	3	103,923	1,277	585.8
4	2	2	120,000	1,596	721.8
5	1	5	134,164	1,720	936.2
6	2	3	146,969	1,989	1,064
7	1	7	158,745	2,180	1,224

測定の結果、７ノードすべてを使ったとき、演算性能 1,224Gflops（1.224Tflops）を達成した。

（４）理論値との比較

測定結果の演算性能を理論ピーク値と比較する。そのためにまず理論ピーク値をノードあたり CPU および GPU について求める。

CPU の演算性能の理論ピーク値は、コアあたり以下の計算式で計算できる。

周波数×同時演算数

最近の PC 向けプロセッサでは、

同時演算数＝SIMD 分×積和同時演算分

＝SIMD 分×２

である。今回使用した CPU（Intel Core i7-8700K）は内臓する６コアすべてブーストしたときの各コアの最大周波数は 4.3GHz である[16),17)]。また、x86_64 アーキテクチャの命令セットでは最近の AVX 命令セットを使用すると SIMD 分は４となる。ただし、今回採用したソフトウェア環境の制約があり、GotoBLAS2 は 2010 年に開発が終了しているため AVX に対応していない。SSE シリーズの命令セットには対応しているので、SIMD 分は２となる。理論値としてどちらを採用するか判断に迷うため、以降では両者を併記する。以上から、理論ピーク値は、

34.40Gflops/コア（AVX 使用時）

17.20Gflops/コア（SSE 使用時）

となり、システムには 42 コアあることから、

1,445Gflops（AVX 使用時）

722.4Gflops（SSE 使用時）

がシステム全体の CPU 分の理論ピーク値となる。

一方、GPU の演算性能の理論ピーク値については、NVIDIA 社の公表値[18)] を使用する。今回使用した GeForce GTX 1060 3GB の倍精度実数演算性能の理論ピーク値は、単精度の場合の公表値 4.0Tflops の 1/32 であり、

125.0Gflops/ボード

である。システム内に７ボードあるので、

875.0Gflops

がシステム全体の GPU 分の理論ピーク値である。

以上から、CPU 分と GPU 分合わせたシステムの演算性能の理論ピーク値は、

2,320Gflops（CPU では AVX 使用）

1,597Gflops（CPU では SSE 使用）

であることが分かる（後者は端数 0.40Gflops が付く）。

実測値 1,224Gflops とこれらの値から、演算性能の実行効率は 52.76%（CPU では AVX 使用を仮定した理論値との比較）もしくは 76.62%（同 SSE）であることが分かった。

５．議論

制作したクラスタマシンは極めて標準的な環境構築を選んだこともあり、些かの問題もなく動作

し、ベンチマーク結果から得られた実行効率はソフトウェアの制約を考慮すれば 76.62%と、文献[11),19)]にある値（それぞれ 83.1%、53.5%）と比較して妥当な範囲に収まった。性能について特筆すべき点はないものの運用しやすい並列計算環境を得られたと言える。

敢えて特徴を挙げるならば、民生用量販機器で構成したクラスタマシンであるので、安価に構築でき、費用に対しては得られた演算性能が高い点であろう。今回、製作で費やした物品費は、表-1 に示した機材の他にディスプレイやケーブル、HDD 複製機といった末端のものまで含めて総額 153 万 9 千円であった。仮に各ノードにおおよそ同等の理論演算性能を持つ市販の計算ワークステーション、例えば D 社のある製品（Intel Xeon W-2135、32GB DDR4-2666 ECC RAM、NVIDIA Quadro P2000 搭載）を使って構成する場合には執筆時点では総額 395 万 8 千円かかる。このように、民生用量販機器を利用したことで大幅な費用削減になっていたことが分かる。この種の狙いを持って制作されたクラスタマシンは多くの制作事例があり、国内では大阪大学の事例[20)]と長崎大学の事例[21)]は特に大規模なものとして知られている。

一方で、やや高価な機材で 1 台のワークステーションを構成した方が、安価な構成を採用したクラスタマシンよりもさらに費用面で効率的に演算性能が上げられる可能性はある。実際、使用部品の市場価格総額が約 75 万円であるが、理論ピーク演算性能が（CPU で SSE 命令セット使用を仮定して）4.001Tflops に達する計算ワークステーションが当研究室にある。このホストは理論ピーク演算性能 1.882Tflops の GPU である NVIDIA GeForce GTX TITAN BLACK を 2 ボード搭載している。（その他、CPU：AMD Threadripper 1950X、メモリ：64GB DDR4-2400 ECC RAM を搭載。）演算性能を hpl-2.0_FERMI_v15 で測定した結果、657.3Gflops を得た（このときのパラメータ値は N=72,000、NB=768、NP=2、P=1、Q=2、CPU_CORES_PER_GPU=8、GOTO_NUM_THREADS=8、CUDA_DGEMM_SPLIT=0.90、CUDA_DTRSM_SPLIT=0.80）。なお、このホストでは GotoBLAS2 は OPTERON 系をターゲットに make した場合動作が不安定であり、NEHALEM ターゲットでは演算性能がやや低く出たため、CORE2 ターゲットで make したものを使用した。演算性能に対してメモリ搭載量が HPL ベンチマークの測定には十分でないため、理論値に比べ測定結果は振るわず、実行効率は 16.43%となったが、それでも 1 台で今回構築したクラスタマシンのおおよそ半分の演算性能を示した。

ただこのホストと今回構築したクラスタマシンを比較して、一概に優劣は付け難く、扱う計算による。まず、科学技術計算の中でも典型的には Linpack が行っているような LU 分解に代表される、計算量がたかだか $O(N^3)$で（計算量の）係数も小さい行列計算を考える。この場合行列サイズが大きいほど演算性能が理論ピーク値に近づく傾向にあり、メモリ容量が十分大きい場合に演算器の実力が発揮できる。1 台に搭載できるメモリの容量は、中価格帯のワークステーションではマザーボードが量販モデルであり 128GB 程度が限度であるため、搭載される強力な演算器の能力と見合わない。上述の Threadripper 搭載のホストでも 128GB が上限である。数 Tflops 以上の性能の演算器に対しては 256GB 程度以上のメモリ容量を用意すべきであり、そうなると（概ね 1 台 200 万円を超える）高価格帯のワークステーション向けの規格のマザーボードが必要である。一方、演算器の性能がそれほど高くないのであれば、メモリ容量が少ない場合でも典型的な行列計算の性能は理論ピーク値に近いものが得られる。したがって、現在の計算機市場においては、クラスタマシン構成の方が、演算器の性能とメモリ容量のバランスが良いノードを並べられ、典型的な行列計算での実行効率を高くしやすい。次に、データ量に対して演算量が多い計算処理を考える。例えば長距離相関が無視できない多体力学系で長時間にわたる時間発展を計算するとか、準指数時間かかる解法しか発見されていない近似最適化問題を解く、といった場合が挙げられる。こういった処理であれば、搭載メモリ容量に制限があっても演算器の性能が高い中価格帯のワークステーション 1 台の方が低価格構成の小規模クラスタよりも適している。

別の視点では、拡張性を見ると低価格構成のクラスタマシンは優れている。一度クラスタマシンを構築すれば、構築後にノードを増やすのは容易であり、PC を調達して HDD を既存ノードからコピーする他にわずかの追加設定で済む（３-（９）で述べた手順を参照)。

また、最初に述べたように、スーパーコンピュータにジョブを投入する前のテストベッドとしては、今回十分な環境を構築できた。表-5 を見ると、ノード数を増やした時の演算性能のスケーリングは線形ではないものの 7 ノードのときに１ノードの６倍程度と、十分である。この環境でスケーリングが認められるプログラムであれば、より大規模なクラスタマシンでもスケールすると予想される。

今回の構成上、唯一、テストベッドとして欠けているものはリソース管理ソフトである。スーパーコンピュータや大規模クラスタマシンでは通常ユーザ数が多く、計算リソースを公平に分配するため、ユーザごとに使用できるリソースを設定し、投入するジョブを管理する仕組みが必要になる。よく使われるリソース管理ソフトとして、TORQUE[22] がある。現状では今回作成したクラスタマシンでは多人数での利用はないと考えられ、導入する意義が薄いが、今後、ジョブ投入の練習のために必要があれば導入を検討したい。

６．おわりに

量販型の PC 機材で構成した７ノードのクラスタマシンを制作し、情報学科サーバ室に設置した。本稿では構築作業を可能な限り詳細に述べ、ベンチマークテストの結果を報告した。Linpack テストの結果は（倍精度）演算性能 1.224Tflops であり、実行効率はソフトウェアの制約を考慮した値で 76.62%であった。並列計算プログラムの実験を気兼ねなく行える計算資源として今後活用していきたい。

謝辞

本研究は、科学研究費補助金 基盤研究（C）（課題番号：18K11344）および崇城大学平成 30 年度研究重点配分予算（超高精度高速量子計算機シミュレータの実証的研究）の支援を受けた。

参考文献

1) I. Cutress, “AMD 16-Core Ryzen 9 3950X: Up to 4.7GHz, 105W, Coming September,” Web news, AnandTech (https://www.anandtech.com), June 11, 2019

2) 後藤弘茂、“AMD や IBM、Arm が「Hot Chips 31」で CPU アーキテクチャを公開”、ニュース記事、PC Watch (https://pc.watch.impress.co.jp)、2019 年 8 月 23 日

3) 理化学研究所計算科学研究センターおよび高度情報科学技術研究機構編著、“スーパーコンピュータ「京」年報 2017-18”、2018 年 12 月

4) 相川 勝、武居 周、“大規模有限要素解析のための並列計算機環境構築と性能評価”、平成 29 年度電気・情報関係学会九州支部連合大会、11-1P-01、p.210 (2017)

5) 中尾 昌広 他、“仮想クラスタの構築と性能評価”、同志社大学理工学研究報告 **50**(4), pp.20-24 (2010)

6) A.O. Kudryavtsev, V.K. Koshelev, and A.I. Avetisyan, “Prospects for Virtualization of High-Performance x64 Systems,” Programming & Computer Software **39**(6), pp.285-294 (2013)

7) 幸谷 智紀、“Vine Linux による PC Cluster の構築”、2003 年 3 月、http://na-inet.jp/na/mpipc.pdf

8) A. Petitet, R.C. Whaley, J. Dongarra, and A. Cleary, “HPL−a portable implementation of the High-Performance Linpack benchmark for distributed-memory computers,” Version 2.3, Dec. 2018, https://www.netlib.org/benchmark/hpl/

9) A. SaiToh, “ZKCM: A C++ library for multiprecision matrix computation with applications in quantum information,” Comput. Phys. Comm. **184**, pp.2005-2020 (2013)

10) J. Dongarra, “Performance of Various Computers Using Standard Linear Equations Software,” Tech. Rep. CS-89-85, Univ. Tennessee, 1989 (updated version: http://www.netlib.org/benchmark/performance.ps)

11) 高橋 大介 他、“T2K 筑波システムにおける Linpack 性能評価”、情報処理学会研究報告 2008-HPC-116 (10), pp.55-60 (2008)
12) A. Fatica, “Accelerating linpack with CUDA on heterogenous clusters,” in the 2nd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-2), 8 Mar. 2009, Washington, DC
13) E. Phillips and M. Fatica, “CUDA Accelerated Linpack,” Version 1.5, 2012, https://developer.nvidia.com/computeworks-developer-exclusive-downloads
14) K. Goto, Texas Advanced Computing Center, “Past Project: GotoBLAS2, ” Version 1.13, Feb. 2010, https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2
15) NVIDIA 社の開発者向けウェブサイトで 2017 年にあった議論の記録を参照：https://devtalk.nvidia.com/default/topic/1027360/cuda-accelerated-linpack-not-running-undefined-symbol-dtrsm/
16) Tom’s Hardware, “Intel Core i7-8700K Review, Oct. 2017, ” https://www.tomshardware.com/reviews/intel-coffee-lake-i7-8700k-cpu,5252.html
17) wccftech, “Intel Coffee Lake Core i7-8700K and Core i5-8600K 6 Core CPUs Leaked, ” Jul. 2017, https://wccftech.com/intel-coffee-lake-core-i7-8700k-core-i5-8600k-6-core-cpu-leak/
18) wccftech, “NVIDIA GeForce GTX 1060 Official Specification and Benchmarks,” Jul. 2016, https://wccftech.com/nvidia-gtx-1060-specificaitons-and-benchmarks-leaked/
19) 遠藤 敏夫、額田 彰、松岡 聡、“異種アクセラレータを持つ TSUBAME スーパーコンピュータの Linpack 評価”、応用数理 **20**(2), pp.29-36 (2010)
20) 菊池誠、時田恵一郎、茶碗谷毅、“待兼山計畫”、日本物理学会 2000 年春の分科会、22aZC-2 、2000 年 9 月 (2000); 次の URL も参照：https://www.cmc.osaka-u.ac.jp/publication/annual-report2000/13-17.html
21) 濱田 剛 他、“GPU を用いたサブペタフロップス高性能計算機システム”、映像情報メディア学会技術報告 32.54(0), pp.17-19 (2008)
22) Adaptive Computing Enterprises, Inc., “TORQUE (Terascale Open-source Resource and Queue manger), ” Version 6.1.2, Feb. 2018, https://github.com/adaptivecomputing/torque