ノード間通信が可能なボランティアコンピューティング

田中 創樹[1]　新城 靖[1]

概要：ボランティアコンピューティングとは，ボランティアが提供したインターネット上の計算機資源を集めて，並列計算機として利用する仕組みである．従来のボランティアコンピューティングでは，参加ノードはプロジェクトサーバと通信し，サーバから送られてきたタスクを実行するが，参加ノード間で通信を行うことはない．そのため，ボランティアコンピューティングに適用できるアプリケーションの種類が制限されていた．また，対話的に並列計算をさせることができなかった．本研究では，ノード間通信が可能なボランティアコンピューティングを実現する．本研究では，ボランティアコンピューティングプラットフォームの BOINC が VM 上でタスクを実行する機能を拡張し，同一プロジェクトのノードの VM を VPN で結び，ノード間通信できるようにする．通信できる範囲をノード内の VM に限定することで，安全性を高める．この環境で，MPI を使用する並列計算ベンチマークの NAS Parallel Benchmarks のプログラムを修正すること無く動作させることができた．また 2 つの対話的アプリケーション（マンデルブロ集合の描画とレイトレーシング）が実行できることを確認した．

1. はじめに

科学技術計算において計算資源を得る方法の 1 つに，ボランティアコンピューティング（Volunteer Computing，以下 VC という）がある．これは，インターネットに接続された世界中の多数のコンピュータを集めて，仮想的な並列コンピュータとして利用できるようにする仕組みである．VC の特徴は単一のスーパーコンピュータやグリッドコンピューティングと異なり，構成する計算機ノードをインターネット上のボランティアが提供する点である．計算資源を提供するボランティアは複数の計算プロジェクトの中から貢献したいものを選んで，基本的に無償で自身の計算機の CPU 時間を提供する．

最近の VC の代表的なプラットフォームに，BOINC[1] がある．地球外知的生命体探索を目的とした SETI@home[2] を含む多くの VC プロジェクトが，現在 BOINC で動いている．現在の BOINC システム全体の計算能力は約 69 万ノードで約 20 PetaFLOPS[*1] に達し，スーパーコンピュータの規模に匹敵する．

これらの従来の VC では，通常の並列コンピュータと異なり，ノード間で通信できないため，適用可能なアプリケーションの種類が制限されるという問題がある．例えば，MPI(Message Passing Interface) を利用するアプリケーションは実行できない．また，対話的に並列計算をさせることができなかった．例えば，ユーザの操作に応じて動的にタスクを生成して並列計算を行い，結果を表示することができなかった．

この問題を解決するため，本研究では，VC において同一プロジェクトに所属するノード間での通信を可能にする（図 1）．これにより，VC で実行可能なアプリケーションの種類を拡大する．この目的を達成するため，本研究では全てのノードを VPN(Virtual Private Network) で接続する．この環境で，MPI を使用する並列計算ベンチマークの Nas Parallel Benchmarks をプログラムを修正すること無く動作させることができた．また GUI 操作に追従して対話的にマンデルブロ集合やレイトレーシングの計算結果を描画するアプリケーションが実行できることを確認した．

ボランティアにとって VC で実行するプログラムと通信相手は未知であるから，セキュリティを確保する仕組みが必須である．そこで，本研究ではアプリケーションを VM(Virtual Machine) で実行し，この VM に VPN ネットワークを割り当てる．そうすることにより，外部からの通信を VM 内に限定し，不正なプログラムによる影響をホスト環境に与えられないようにする．

2. VM と VPN による並列計算環境の構築

2.1 VM を用いる BOINC

本研究では，BOINC の仕組みを拡張して，ノード間通信が可能な VC を実現する．BOINC[1] はオープンソース

[1] 筑波大学

[*1] 2017 年 10 月 23 日時点の 24 時間平均．
`https://boinc.berkeley.edu` より．

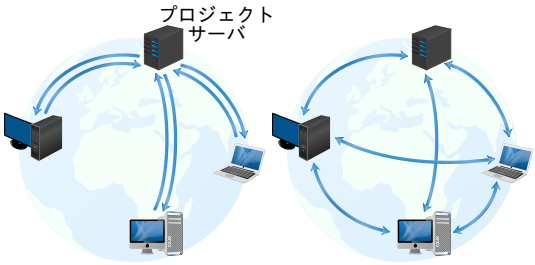

図 1　従来の VC においての通信 (左) と，ノード間通信できる VC(右)

の VC プラットフォームである．並列計算をして欲しいプロジェクトは，タスクを配布するためのサーバ（プロジェクトサーバ）を用意し，BOINC ポータルサイトに登録する．ボランティアは BOINC クライアントをインストールして BOINC ポータルサイトに接続する．そこで参加するプロジェクトを選択すると，プロジェクトサーバから計算全体を細かく分割したタスクが配布される．参加ノードはこれを解いて結果をプロジェクトサーバに返却する．プロジェクトサーバは全体の結果を集計し，目的の計算を完了させる．

異なる OS のクライアントで同一のアプリケーションを実行できるようにし，開発者の負担を減らすため，BOINC は配布タスクを VM(VirtualBox) で実行する仕組みを持っている [13][14]．この機能を使う場合，プロジェクトサーバはタスク本体に加え，下記をクライアントに配布する．

(1) VM の起動イメージ
(2) VM の起動設定ファイル (vbox_job.xml)
(3) VM を立ち上げるプログラム (vboxwrapper)

この仕組みでは，BOINC クライアントは，vboxwrapper を実行し VM を立ち上げる．VM が起動すると，VM 内でプロジェクトから配布されたプログラムが起動し，ホストに保存されたタスクを VM の共有フォルダ機能でアクセスし実行する．

VM を用いる BOINC では VM のネットワークは起動設定ファイルに従い，ホストの NIC(Network Interface Card) にブリッジで接続される．その場合，vboxwrapper がホスト環境からアクティブな NIC を探し，最初に見つかった NIC に接続する．

2.2　VPN による VM の接続

本研究では，2.1 節で述べた BOINC の仕組みを拡張し，VM 内では他のノードと VPN で通信できるようにする．そのために，プロジェクトの主催者は，VPN サーバおよび DHCP サーバを用意する（図 2）．各ボランティアのホストでは，VPN クライアントを実行し，プロジェクトの VPN サーバへ接続する．そして VM のネットワークを，VPN に接続する．これにより，全てのボランティアの VM が同一の VPN サーバに接続され，相互に通信可能になる．DHCP サーバは，VPN に接続された VM に IP アドレスを割り当てる．

本研究では，2.1 節で述べた VM を用いる BOINC を次のように改変した．

- VPN の接続設定を，プロジェクトサーバからタスクと共に配布する．
- VM の立ち上げ時に，VPN サーバへ接続する．
- VM の仮想 NIC をブリッジで VPN の仮想 NIC に接続する．

ボランティアのホストでは，安全のために，他ノードから自身へは，通信ができないようにしたい．自身の環境に通信できてしまうと，不正にアクセスされたり，攻撃されるおそれがあるからである．そこで本研究では，VPN への接続時は，ホスト側の VPN の仮想 NIC に IP アドレスを付けないようにした．そして VM 内で DHCP クライアントを立ち上げ，プロジェクトサーバが管理する DHCP サーバから IP アドレスを割り当てるようにした（図 2）．こうすれば，他のボランティアやプロジェクトサーバと通信できる範囲は VM 内に限られ，自身のホスト環境に影響を与えられるのを防げる．

2.3　SoftEther VPN による VPN の構築

本研究ではプロジェクトのノードを VPN で結ぶために SoftEther VPN[3] を利用することにした．この VPN は，インターネット上のノードを仮想的な L2 スイッチに接続する．ノード同士は NAT 無しで通信可能になる．これは，並列計算に加わるノードの管理を容易にする．

SoftEther VPN は IP アドレスを割り当てずに仮想 L2 スイッチに接続できる．そのため，2.2 節で述べたように IP アドレスを VM 側 NIC だけに付け，安全性を高めることができる．

2.4　NFS サーバによるプログラムとデータの共有

本研究では，ボランティアの各ノードで同一のプログラムを参照できるようにする．本研究では，このためにプロジェクトサーバで NFS サーバを実行する．プロジェクトサーバが，実行するプログラムを共有ディレクトリに配置し，これをボランティアの VM が VPN 上でマウントする．この方法により，プログラムだけでなく，データの配布や計算結果の返却も共有ディレクトリを通じて行えるようになる．

2.5　Docker コンテナによる実行環境の構築

VC では，ボランティアの各ノードで，ライブラリのパス，バージョンを揃えたいという要求がある．この問題を解決するため，本研究では VM に加えて Docker を利用することにした．これにより，VM イメージには汎用性を持たせ小さく保ちつつ，簡単に実行環境を揃えられるように

なる．タスクを Docker コンテナとして配布する仕組みは，boinc-server-docker[5] を使用した．

従来の boinc-server-docker では Docker コンテナを立ち上げると，コンテナの NIC がホスト内部に作成された docker0 というブリッジに接続され，NAT(Network Address Translation) により，ホストの NIC に接続される．この方法では，VPN 内の他のノードと同一ネットワークに置かれないという問題がある．そこで本研究では，コンテナのネットワークを VM と直結させ，コンテナ間が直接 VM の IP アドレスで通信できるようにした．

2.6 プロジェクト管理者がやること

プロジェクト管理者はタスクの配布前に NFS サーバ，DHCP サーバ，VPN サーバを設定しておく．また各クライアントに配布する Docker コンテナを用意しておく．この Docker コンテナは，立ち上がり時に，NFS サーバの共有ディレクトリをマウントするように，Docker ファイルを設定しておく．また，コンテナ起動時に DHCP クライアントにより IP アドレス割り当てを要求するコマンドが実行されるようにしておく．そして，boinc-server-docker を使用して，このコンテナをタスクとしてプロジェクトに登録する．

2.7 各ボランティアがやること

各ボランティアはタスクの実行前に Softether VPN のクライアントをあらかじめ立ち上げておく．ただし，VPN 接続はこの時点では行わない．VPN の設定と接続は後に vboxwrapper が行うからである．また VM の VirtualBox をインストールしておく．以後は通常の実行手順と同様である．つまり，BOINC クライアントを実行し，支援するプロジェクトを追加して，タスクが配布されるのを待つ．タスクを受け取ると，vboxwrapper は VM を立ち上げ，その VM の NIC とプロジェクトの VPN の NIC をブリッジで接続する．そして，この VM の中でコンテナが立ち上がり，IP アドレスの割り当てと共有ディレクトリへの接続が自動で行われる．そして，コンテナの中でタスクが処理される．

3. MPI アプリケーションの実行

3.1 目的

MPI は複数のノードを用いた並列計算でノード間の同期や通信に使用される標準である．MPI は問題に適したトポロジを作成したり，各ノードからの結果を集約する等，便利な機能を備えており，科学技術計算で広く使われている．研究者は，MPI を用いて LAN で動作するプログラムを開発して実行する．研究者は，計算機資源が不足してきた場合，VC を利用したいと考える．

従来の VC はノード間で通信できないので，LAN で動作

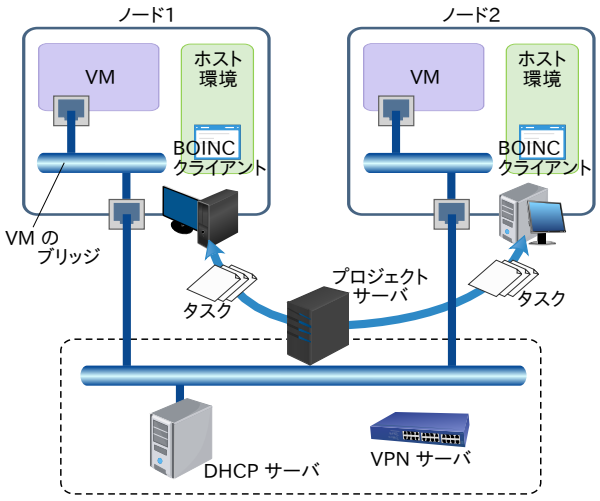

図 2 VPN と VM との接続

している MPI のプログラムを VC で利用することはできない．そのため，研究者は，MPI のプログラムを VC で実行できるように，修正する必要があった．この修正は容易ではない．また，動的にタスクを生成するような計算は，従来の VC にはまったく適していないと考えられてきた．

しかし，MPI アプリケーションを VC で動かせると，MPI を使用する並列計算プログラムを VC 用に書き直さなくてよくなる．これは，研究者にとってアプリケーション開発が容易になるという，大きな利点がある．また，MPI を使って動的にタスクを生成するようなアプリケーションであったとしても，VC で実行可能になる．

3.2 ノードの IP アドレスの管理

本研究では，ボランティアが提供している計算ノードは VPN で結ばれているので，基本的には LAN で動作する MPI の並列アプリケーションがそのまま動作する．しかし，VC として使いやすいものにするためには，いくつかの課題がある．まず，計算ノードの IP アドレスの管理に課題がある．VC では，計算の途中で新たにスレーブが参加したり，離脱することがある．MPI のマスタでは，利用可能なスレーブのリストが必要である．

本研究では MPI のマスタノードが並列計算に参加させるスレーブノードの IP アドレスを DHCP の仕組みを使って把握できるようにする．ボランティアのノードが新しく参加すると，IP アドレス割当てのため，プロジェクトサーバで動作している DHCP サーバに IP アドレス割当要求を行う．本研究で使用する DHCP サーバの ISC DHCP は，IP アドレスの割当と開放のタイミングで，外部コマンドを実行できる．その際に，割当 IP アドレスと，クライアントの MAC(Media Access Control) アドレスを引数に与えることができる．この機能を利用して，接続中のノードを管理する．

プロジェクトサーバはデータベース (MariaDB) を持つ．

表 1　実験に使用した PC

ノード	CPU	メモリ
マスタ	Intel Core i7 6700	DDR4 8GB
スレーブ 1	Intel Core i5 750	DDR3 12GB
スレーブ 2	Intel Core i7 7700K	DDR4 16GB

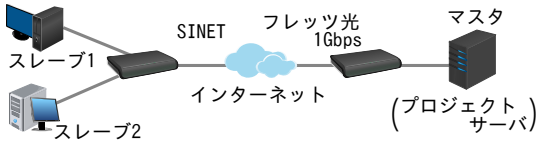

図 3　ノード間のネットワーク

そしてスレーブノード管理用のテーブルを用意しておく．IP アドレス割当時に DHCP サーバから呼ばれるプログラムは，引数で受けた IP アドレスと MAC アドレスの組を，このテーブルに追加する．スレーブノードが終了して IP アドレスが開放される時に実行するプログラムは，テーブルからこのノードが持っていた IP アドレスの要素を削除する．

MPI では，マスタノードが並列計算に参加させるノードを hostfile により指定できる．本研究では，IP アドレスをデータベースのノード管理テーブルに追加・削除した後，このテーブルの IP アドレスの情報を元に，hostfile を自動的に更新する．

3.3　SSH 鍵の設定

MPI で並列プログラムを実行する場合，マスタからはスレーブへ，スレーブからはマスタ，および他のスレーブへ SSH 接続できるようにする必要がある．本研究では，マスタとスレーブのコンテナ環境を規定する Dockerfile に，これらのノード間で SSH 接続できるように設定する．具体的に，公開鍵認証のための秘密鍵の作成と，対応する公開鍵の交換が既に済んでいるようにした．これにより，各ノードはプロジェクト内の任意のノードにパスワードの入力や警告メッセージの確認無しで，SSH 接続できる状態で立ち上がる．

3.4　NAS Parallel Benchmarks の実行

作成した MPI 実行環境で MPI のアプリケーションが動作することを確かめるため，MPI を使う NAS Parallel Benchmarks を使用した．このベンチマークは 5 つの計算カーネル (IS, EP, CG, MG, FT) と 3 つの擬似アプリケーション (BT, SP, LU) からなり，それらは異なったパタンのメモリアクセスと通信を行う．これらの全てのプログラムを，変更すること無く，本環境で実行できることを確認した．

これらの並列プログラムをマスタ 1 台で実行した場合と，VC でスレーブ 2 台を加えて計 3 台で実行した時で，実行時間を測定した．実験環境の PC の性能を表 1 に，ネットワークの構成を図 3 に示す．この構成で，プロジェクトサーバでは VPN サーバと MPI マスタが動作している．MPI スレーブ間の通信も，この VPN サーバを介して行われる．表 2 に，ping を用いて測定したレイテンシを示す．このように，マスタとスレーブの間では，レイテンシは，13 ミリ秒であったが，スレーブ間では，その倍になった．図 4 に，ノード間のスループットを示す．マスタとスレーブの間では，順方向は 100M bps 以上と高速であったが，逆方向は，20M bps 程度しか得られなかった．スレーブ間でも，15M bps であった．ただし，これらのスループットは日や時間帯によって大きく変動する．また，以下 NAS Parallel Benchmarks の実験を行った日とは別の日に測定したものである．

表 2　各ノードへの ping レイテンシ (ミリ秒)

	レイテンシ
マスタ - スレーブ 1	13.2 (0.8)
マスタ - スレーブ 2	13.3 (0.9)
スレーブ 1 - スレーブ 2	25.7 (0.8)

(括弧内は標準偏差)

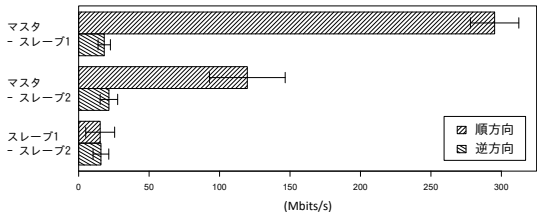

図 4　ノード間のスループット

この実験では 2.1 節で述べた，VM を用いる BOINC における設定ファイル vbox_job.xml で指定する VM のメモリサイズは 10GB に設定し，BOINC クライアント側では CPU とメモリ使用に制限をかけないようにした．プログラムは 16 プロセスで実行し，スレーブを含めて実行する時は，各スレーブに 4 プロセスずつ分配した．

NAS Parallel Benchmarks の実行結果を表 3 に示す．EP(Embarrasingly Parallel) と LU(Lower-Upper Gauss-Seidel solver) は，マスタ 1 台で行うとそれぞれ 29 秒，および，228 秒掛かったが，VC で 3 台で実行すると実行時間がそれぞれ 18 秒，および，108 秒と短くなった．一方で，CG(Conjugate Gradient) と FT(discrete 3D fast Fourier Transform) では，実行時間が逆に長くなった．

VC で実行時に測定した各ノードのスループットを表 4 に示す．このように，単位時間あたりの通信量は EP と LU が比較的小さいが，FT では大きいことがわかる．並列計算の通信にインターネットを用いるために，通信帯域がボトルネックになって遅くなっていると考えられる．この結果から，EP と LU ではインターネットを超えた通信の影響が小さかったが，CG と FT では，大きかったと思われる．

表 3　各プログラムの実行時間 (秒)

プログラム（クラス）	1 台	VC
IS(C)	39	118
EP(C)	30	19
CG(B)	186	208
MG(C)	46	59
FT(A)	9	26
BT(B)	135	258
SP(A)	217	169
LU(B)	229	103

表 4　各プログラムの実行時の単純平均スループット (Mbits/s)

		マスタ	スレーブ 1	スレーブ 2
EP	受信	0.021	0.015	0.012
	送信	0.022	0.016	0.015
LU	受信	24	24	47
	送信	24	25	51
FT	受信	87	65	66
	送信	88	72	73

3.5　評価

MPI を使用したノード間の並列プログラムが，本環境で修正すること無く動作したが，実行時間が遅くなるプログラムもあった．インターネットを介した通信では，利用可能な帯域や通信遅延が刻々と変化する．計算ノードがクラッシュすることもある．LAN で動作するプログラムと異なり，VC ではこのような点を考慮する必要がある．アプリケーションにより，通信パタンもさまざまである．アプリケーションによっては，全体の通信量はそれほど多くはないが，特定のタイミングに集中することがある．通信量，通信回数，および，通信パタンがどれだけ性能に影響してくるか調べることは，今後の課題である．また，ノードが（意図的に）間違った結果を送ってくることもありうる．従来の VC では，同一のタスクを複数のボランティアに送り，結果を比較して検算する方法がある．MPI を使用した並列計算アプリケーションにおいても，この問題を解決する必要がある．

4.　対話型アプリケーションの実行

4.1　目的

従来の VC における科学技術計算はバッチ処理によるものが普通である．実行するプログラムや計算に使用するデータをあらかじめひとまとめにしておいて，ボランティアに配布する．しかし，ユーザの操作に応じて，計算に用いるパラメータの値を随時変えて，結果を逐一確認したいという要求もある．研究者は LAN で科学技術計算を行う時には，そのようなことが簡単にできる．また，IPython や Jupyter Notebook[9] のような対話的な計算環境が普及し，科学技術計算に用いられるようになってきており，対話的に大規模計算を行いたいという要求が増大していくと想定する．本章では本研究で作成した VC 環境で，このような対話型並列計算アプリケーションが実行できることを示す．

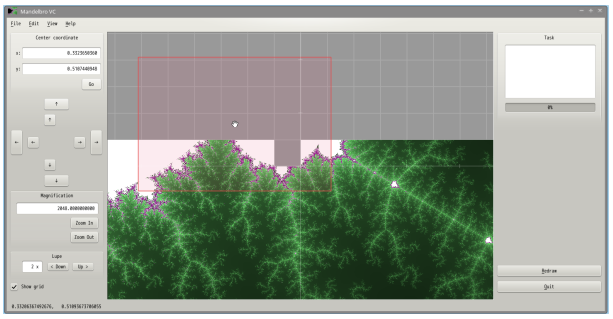

図 5　VC を使ったマンデルブロ集合の高速描画アプリケーション (MandelbroVC)

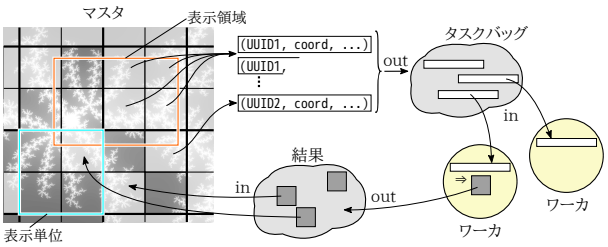

図 6　Linda によるマスタ・ワーカモデル

4.2　マンデルブロ集合の描画アプリケーション

本環境を利用する対話型アプリケーションの例として，マンデルブロ集合をブラウジングするプログラムを作成した[*1]．このプログラムは，マウスのドラッグによって描画する範囲を動かしたり，ダブルクリックやホイール操作によって拡大・縮小することができる．C++と Qt4 で作成され，Linux 上で動作する．そのスクリーンショットを図 5 に示す．ウィンドウの中央は，マンデルブロ集合の描画領域である．ウィンドウの左側には，移動と拡大，倍率切り替えのボタンがあり，右側にはタスクの進捗率を表すプログレスバーと再描画ボタンがある．この図は，未描画領域（中央の表示領域内の灰色部分）が次々と計算して得られた画像で埋め尽くされていく最中の様子を表す．

このプログラムは，マスタ・ワーカモデルに基づいて設計されている．マスタがタスクを生成する．マスタが生成したタスクをワーカは計算し，結果をマスタに返す．マスタは返された結果を利用する．マンデルブロ集合の画像の計算は，画素毎に独立して行えるから，ワーカ同士で通信が発生せず，この方式を自然に適用できる．一枚の画像を得るだけであれば従来の VC を使うこともできるが，このプログラムのように，実行時の操作で動的にタスクを生成して絵を変えるような使い方はできなかった．

タスク生成は，画面を一定の大きさで区切った表示単位

*1　MandelbroVC (`https://github.com/tsoju/MandelbroVC`)

```
LINDA::TYPE::Integer div,pos,w,h;
LINDA::TYPE::Double x_s,x_e,y_s,y_e;
LINDA::TYPE::String id(ti.uuid_s);
/* div_l_x(y): 複素平面上での 1タスクの幅 (高さ) */
/* div_v(h): 表示単位の縦 (横)方向の分割数 */
double div_l_x = (ti.crd.x_e - ti.crd.x_s) /
                 (double)ti.div_h;
double div_l_y = (ti.crd.y_e - ti.crd.y_s) /
                 (double)ti.div_v;
w = ti.dw;  /* 横方向ピクセル数 */
h = ti.dh;  /* 縦方向ピクセル数 */
for(int i=0; i<ti.div_v; i++){
  for(int j=0; j<ti.div_h; j++){
    pos = (i<<8) | j; /* pos: 表示単位内での位置 */
    /* [x(y)_s, x(y)_e]: 計算する領域 */
    x_s = ti.crd.x_s + div_l_x * j;
    y_s = ti.crd.y_s + div_l_y * i;
    x_e = x_s() + div_l_x;
    y_e = y_s() + div_l_y;

    linda_server->out(&id,&pos,&x_s,&y_s,
                      &x_e,&y_e,&w,&h);
  }}
```

図 **7**　マスタがタスクをタスクバッグに格納するコード

で行う．図 5 のように，表示領域が動かされ，領域内に未計算の表示単位を含むようになったら，この表示単位を分割して計算タスクを生成する．このタスクをワーカに配布して計算させ，結果を受け取って，描画に使用する．

同一の表示単位から生まれたタスクは，一つのタスクグループを形成し，グループを識別するための ID(UUID) が与えられる．この ID によってそれぞれのタスクの所属グループを判別している．

タスクの配布機構として，本研究では，まず Linda を使用した [10]．Linda はタプル空間へのいくつかの単純な操作 (`out`，`in`，`rd`，`eval`) のみで，並列プログラムに通信や同期機能を提供するモデルである．したがって，本研究がターゲットとしている研究者でも比較的簡単に Linda を使って並列プログラムを記述できると思われる．本研究では Linda の C++実装である CppLinda[12] を使用した．

本プログラムのマスタは生じた未描画の表示単位を分割してタスクとしてタプル空間上に設置したタスクバッグに格納する (図 6)．タスクを表すタプルは，次の要素を含む．

(1) 所属タスクグループを示す ID
(2) 表示単位内での位置
(3) 描画する複素平面上の領域を示す値

ワーカはタスクバッグからタスクを取り出して計算し，結果をタプル空間に加える．マスタはこの結果を取り出して描画に使用する．

タスクの受け渡し部分のコードを図 7 に示す．図 7 でマスタが表示単位からタスクを生成してタスクバッグに格納している．ループ内でタスクを作るタプルを作成し，`out` 命令でこれをタプル空間に追加する．

```
while(1){
  id.toFormal();  pos.toFormal(); x_s.toFormal();
  y_s.toFormal(); x_e.toFormal(); y_e.toFormal();
  w.toFormal();   h.toFormal();

  linda_server->in(&id,&pos,&x_s,&y_s,
                   &x_e,&y_e,&w,&h);

  if(id().compare("end") == 0) /* 終了の指示受信 */
    break;
  res = new int[w() * h()];
  for(i=0; i<h(); i++){
    for(j=0; j<w(); j++){
      /* (略) 計算して結果を配列res[]に入れる */
    }}
  base64 = g_base64_encode(
    (unsigned char *)res,sizeof(int) * w() * h());
  result_string = new std::string(base64);
  result = *result_string;

  linda_server->out(&id,&pos,&result);

  delete result_string;
  g_free(base64);
  delete[] res;
}
```

図 **8**　ワーカがタスクを取り出し，計算して結果を返すコード

```
id = uuid_s; /* 受信タスクグループID で actual に */
for(int i=0; i<(div_v * div_h); i++){
  pos.toFormal(); result.toFormal();

  linda_server->in(&id,&pos,&result);

  res_data = g_base64_decode(result().c_str(),
                             &out_len);
  update_rectangle(pos()&0xFF,pos()>>8,
                   res_data);
  /* (略) 画像作成と描画スレッドへの更新の通知 */
  g_free(res_data);
}
```

図 **9**　マスタが結果を受け取り，描画を更新するコード

図 8 にワーカが結果を受け取り，計算して結果を返すコードを示す．冒頭の `in` 命令は引数のテンプレートとマッチングするタプルをタプル空間から取り出す．この例のように，テンプレートに formal 変数を使用すれば，型と一致するタプルにマッチングする．取り出したタスクを計算し結果は int 型の配列に得られる．これをタプルに格納できる型付きの値とするため，ここでは Base64 により文字列にエンコードし，これを `out` 命令でタプル空間に加える．

図 9 に，マスタが結果を取り出して描画に使用するコードを示す．`in` 操作に渡すテンプレートで，タスクグループ ID だけは，受信する対象のタスク ID を設定し，actual 変数にしている．actual 変数は型と値が同じものにマッチするので，このように受信したい結果を指定して待つことが

できる．そして結果を元の int 型配列にデコードし，これを用いて画像を作成し表示する．

本研究で実装した VC 環境においてアプリケーションの実行が高速化されるかは，計算量，通信量，通信パタン，ノードの性能，利用可能なネットワークの帯域，および，通信遅延による．このプログラムを，3 章と同じ環境で実行した所，単一ノードで実行したよりも高速に実行できた．このように，Linda を使用して容易に VC で実行できる並列プログラムの開発を行えることが確認できた．この並列プログラムはワーカの性能が異なっても，早く結果を返したワーカから直ぐに次の仕事に取り掛かるので，自動的に負荷分散される．また Linda ではワーカ側からの動的なタスク生成も可能である．これにより，ワーカは受け取ったタスクが大きすぎると判断した場合，タスクをさらに分割するということができるが，このプログラムではまだこの機能は実装されていない．

現在は計算または通信の速度が極端に遅いワーカがあった場合，そのワーカの担当する領域だけ長い間欠けてしまう．また，ワーカが間違った結果を返した場合も，それを有効としてしまう．これらの問題に対処する方法として，次のようなことが考えられる．

- 同一のタスクを複数のワーカに送る．
- 早く計算が完了したワーカの結果を使用する．
- 異なるワーカからの結果が一致するか確かめる．

4.3 Python による科学技術計算

本研究がターゲットとしている研究者は，Python による科学技術計算のアプリケーションを開発することを想定している．Python は Numpy パッケージにより高速な配列演算ができ，特に速度が重要な部分を，Cython[11] を使って C で実装することもできるので，インタプリタ式でありながら，科学技術計算にも利用できる速度を持たせられる．Python からは Matplotlib[8] という，様々な種類のグラフを，カラーマップやアニメーションを用いてわかりやすく表示するライブラリが簡単に利用できる．これらを Jupyter Notebook[9] と組み合わせれば，結果や図の出力を，Web ブラウザ上で対話的に行える．

このように，Python を科学技術計算に用いる環境が，ますます整備されてきている．本節では，そのような科学技術計算の例として，Python によるレイトレーシングが，本研究で構築した VC 環境で動作することを確認する．

4.3.1 Python によるレイトレーシング

本研究では Python で作られたレイトレーシングエンジン [16] を改変し，本 VC 環境で実行した．その実行結果のスクリーンショットを図 10 に示す．図 10 に示した例では，平面上の 3 つの球をレンダリングしている．本研究では，元のプログラムを改変し，視点となるカメラの位置をスライドバーにより対話的に変位できるようにした．また

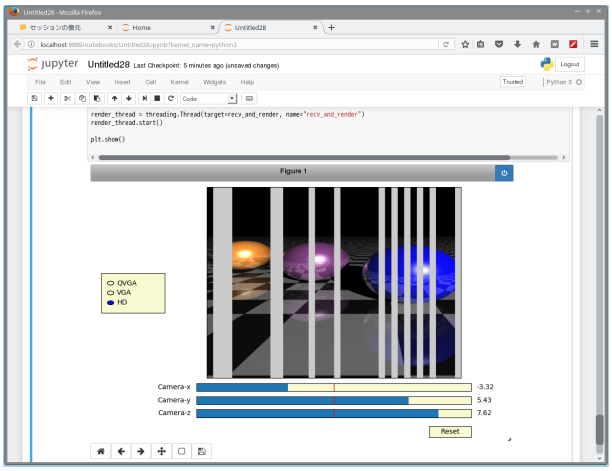

図 **10**　レイトレーシングへの適用

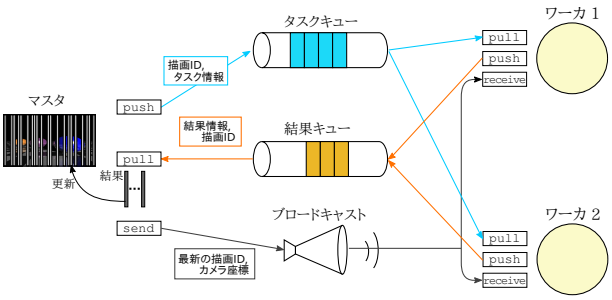

図 **11**　ZeroMQ を使ったタスクのやりとり

解像度を 3 段階 (QVGA, VGA, HD) で切り替えられるようにした．そして，このレンダリングをマスタ・ワーカ方式で並列計算できるようにした．

このプログラムは，ZeroMQ[15] を用いて，次の 3 つのキューを作成する（図 11）．

- マスタからワーカへタスクを送るもの（ZeroMQ の push-pull ソケット）
- ワーカからマスタへ結果を送るもの（ZeroMQ の push-pull ソケット）
- マスタからワーカへ現在計算中の描画 ID を伝えるもの（ZeroMQ の pub-sub ソケット）

本研究で並列化するために加えたコードのうち，重要な部分を図 12，および，図 13 に示す．

使用したレイトレーシングエンジンでは各画素の計算を完全に独立に行える．図 12 で関数 `producer` は画像全体を横方向に分割した短冊型の領域をタスクとしてキューに登録している．`send_json()` は，キューに要素を追加する ZeroMQ の関数である．`recv_json()` は，キューから要素を取り出す ZeroMQ の関数である．`send_array()` は，Numpy の配列データをキューに追加するために用意した関数である．単位タスクあたりの横幅は粒度として変更できるようになっている．

図 13 の関数 `recv_and_render` は，ワーカがキューからタスクを受け取って計算し，結果をマスタの結果用のキュー

```
1  def producer():
2    tlist = []
3    for i, s in enumerate(task):
4      tlist.append([i,s])
5    shuffle(tlist)
6    for i in range(0,max_volunteer):
7      tlist.insert(0,'thanks')
8    for i in range(0,len(task)):
9      e = tlist.pop()
10     sock.send_json({'idx' : e[0], 'from' : e[1], '
        to' : e[1]+ryudo-1, 'ryudo' : ryudo, '
        render_id' : render_id})
11   while tlist != []:
12     e = tlist.pop()
13     sock.send_json(e)
14
15 def recv_and_render():
16   while True:
17     result = recv_array(res_sock)
18     if result == None:
19       continue
20     data = result['data']
21     idx = result['idx']
22     if render_id != result['render_id']:
23       continue
24     last_idx = len(task)-1
25     consumer_id = result['consumer_id']
26     if idx == last_idx:
27       last_w = w - ryudo * last_idx
28       img[:,idx*ryudo:idx*ryudo+last_w,:] = data
        [:,0:last_w:]
29     else:
30       img[:,idx*ryudo:idx*ryudo+ryudo,:] = data
        [:,0:ryudo:]
31     im.set_data(img)
32     fig.canvas.draw_idle()
33     fig.canvas.flush_events()
34
35 render_thread = threading.Thread(target=
       recv_and_render, name="recv_and_render")
36 render_thread.start()
```

図 **12**　マスタのタスク送出と，結果の受信・描画

に格納する関数である．マスタはワーカから結果を受け取り次第，画面上のレンダリング出力を更新する．

計算の最中に，ユーザの操作によりマスタがカメラ座標等のパラメータを変えたら，描画中のイメージは即座に無効にして，新しい結果の描画が開始されるようにした．ワーカが無効になったタスクのために無駄に計算することが無いように，マスタは ZeroMQ のブロードキャストソケット（pub-sub ソケット）で最新の描画 ID(UUID) をスレーブに配信している．スレーブは，受け取ったタスクが最新の描画 ID でなかったら，そのタスクは計算せずに破棄する（図 13 - 7 行目）．

このプログラムを，3 章と類似の環境で実行した所，単一ノードで実行したよりも高速に実行できた．初期設定のままワーカ 1 台で HD 画質のレンダリングが完了するまでの時間を測ると，198 秒だった．同じプログラムを，VC

```
1  def consumer():
2    while True:
3      recv_msg = sock.recv_json()
4      if recv_msg == 'thanks':
5        continue
6      if recv_msg['render_id'] != render_id:
7        continue
8      img = do_work(recv_msg['idx'],
9                    recv_msg['from'],
10                   recv_msg['to'],
11                   recv_msg['ryudo'])
12     send_array(s_socket, img, consumer_id,
13                recv_msg['idx'],
14                recv_msg['render_id'])
```

図 **13**　ワーカがタスクを受け取り，結果を返す

でプロジェクトサーバを含めて 3 つのワーカで実行した場合，93 秒であった．

このプログラムでは 1 タスク当たりの計算時間は平均で 4.4 秒であった．粒度は 32 pixel に設定してあり，タスク自体のサイズは約 90 byte であった．1 タスク分に相当する短冊型の画像の大きさは 720×32 pixel で，データサイズは 67.5 KB であった．表 2 の値を用いて，タスクの配布と結果の収集時間を見積もると，通信時間は約 100 ミリ秒になる．1 タスクあたりの計算時間に対する通信時間の割合が小さいので，本実験ではインターネットを超えた VPN による通信は実行時間にほとんど影響しなかった．

3 章の MPI を使った実験では，インターネットの通信速度がボトルネックになり実行時間が長くなる場合があったが，今回このような性能低下は目立たなかった．それは，本プログラムの下記の特徴による．

- マスタ・ワーカ方式であり，手の空いたワーカがすぐに次のタスクに取り掛かれる．
- タスクの配布，および，結果返却時の通信時間が，計算時間と比較して小さい．

5. 関連研究

SETI@home[2] は 1999 年に公開された VC プロジェクトで，アレシボ天文台が受信した電波から，自然由来ではないと考えられるものを発見することで，地球外知的生命体を見つけようとする試みである．現在までに地球外知的生命体の発見には至っていないが，このプロジェクトはボランティアコンピューティングの実用性を実証した．本研究の手法を用いれば，タスクの配布と結果の返却を NFS サーバが提供する共有ディレクトリを介して行える．

文献 [4] は，ノード同士で通信が発生する VC 環境において動的な参加と離脱を考慮したノード管理構造を提案し，クラスタ間で処理性能を調整する手法を述べている．シミュレーションにより所属ノードの少ないクラスタで性能が低下する問題を解決する手法を示している．しかしノード間で通信する VC の実装は行われていない．本研究は，

VPN と VM を用いてノード間通信可能な VC を実装する．

6. おわりに

この論文では，ノード間通信が可能な VC について述べた．本研究では，VM を用いる BOINC を拡張して SoftEther VPN を用いて VM を接続し，並列計算環境を構築する．本研究で実現した並列計算環境において MPI を用いる NAS Parallel Benchmarks に含まれる 8 つのプログラムを，修正すること無く実行できた．そして，そのうち 2 つのプログラムは高速化できた．また，VC を使用して対話的なアプリケーションを並列に実行できることを確認した．具体的には，マンデルブロ集合の描画と，レイトレーシングのプログラムが単一ノードによる実行より高速に動作した．このことから，このようなタイプの対話的なアプリケーションでは，VC が有効に利用できることを示した．

この論文では，マスタ・ワーカモデルに基づく並列アプリケーションについて述べた．今後の課題は，それ以外のノード間通信可能な VC に向いた並列計算のタイプを見つけることである．VC では計算途中にノードが離脱する場合があるが，現在は考慮していない．ノードの途中離脱による全体への影響を抑える方法の実現も，今後の課題である．

参考文献

[1] D. P. Anderson: *BOINC: a system for public-resource computing and storage*, Fifth IEEE/ACM International Workshop on Grid Computing, pp.4-10 (2004).

[2] SETI@home: https://setiathome.berkeley.edu/. accessed: 2017/7/9.

[3] 登 大遊, 新城 靖, 佐藤 聡 : SoftEther VPN Server: マルチプロトコル対応のクロスプラットフォームなオープンソース VPN サーバ, ソフトウェア科学会 コンピュータソフトウェア, Vol.32, No.4, pp.3-30 (2015).

[4] 菅原 雅也, 福士 将, 堀口 進 : ボランティアコンピューティング環境における動的クラスタ再構成法, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol.2008, No.19 (2008-HPC-114), pp.67-72 (2008).

[5] Marius Millea: A Docker multi-container application that runs a BOINC server, https://github.com/marius311/boinc-server-docker. accessed: 2017/7/9.

[6] Victor Marmol, Rohit Jnagal, and Tim Hockin: *Networking in containers and container clusters*, Proceedings of netdev 0.1 (2015).

[7] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande: *Folding@home: Lessons from eight years of volunteer distributed computing*, IEEE International Symposium on Parallel Distributed Processing, pp.1-8 (2009).

[8] J. D. Hunter: *Matplotlib: A 2D Graphics Environment*, IEEE Computing in Science & Engineering, Vol.9, No.3, pp.90-95 (2007).

[9] Adam A. Smith: *Teaching Computer Science to Biologists and Chemists, Using Jupyter Notebooks: Tutorial Presentation*, J. Comput. Sci. Coll, Vol.32, No.1, pp.126-128 (2016).

[10] Nicholas Carriero and David Gelernter: *How to Write Parallel Programs: A Guide to the Perplexed*, ACM Computing Surveys, Vol.21, No.3, pp.323-357 (1989).

[11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn and K. Smith: *Cython: The Best of Both Worlds*, IEEE Computing in Science & Engineering, Vol.13, No.2, pp.31-39 (2010).

[12] T. A. Sluga: *Modern C++ implementation of the LINDA coordination language for distributed applications*, Bachelor-Thesis, FH Hannover, 2007.

[13] G. A. McGilvary, A. Barker, A. Lloyd and M. Atkinson: *V-BOINC: The Virtualization of BOINC*, 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp.285-293 (2013).

[14] BOINC VboxApps, http://boinc.berkeley.edu/trac/wiki/VboxApps. accessed: 2017/11/6.

[15] zeromq Distributed Messaging, http://zeromq.org/. accessed: 2017/11/7.

[16] Cyrille Rossant: *Very simple ray tracing engine in (almost) pure Python*, https://gist.github.com/rossant/6046463. accessed: 2017/10/23.