Vol.2011-DPS-147 No.17
Vol.2011-MBL-58 No.17
2011/6/3

GPGPUによる無線ネットワークシミュレータJiST/SWANSの高速化

石原　進[†1]　中島和樹[†2]

無線アドホックネットワークのシミュレーションにおいて，各端末が定期的に送信するビーコンも含めたトラフィックの影響も含めて評価するためには，膨大なノード間のパケットの到達性判定処理が必要となる．これらの到達性判定処理には高い並列性がある．近年 GPU (Graphic Processing Units) を汎用の並列計算に用いるGPGPU(General-purpose computing on GPU) が脚光を浴びているものの，無線ネットワークシミュレータでの利用事例は少ない．筆者らは，Java ベースの無線ネットワークシミュレータ JiST/SWANS におけるパケット到達性判定処理を GPGPU により並列処理する方法を設計・実装し，実験によりその効果を確かめた．JiST/SWANS がもつ最も単純なノード管理方式 LinearList にたいして GPGPU による並列化を導入した結果，ノード数 500 台の場合で約 3.4 倍，ノード数 1000 台の場合に約 4.9 倍の速度向上を確認できた．また，JiST/SWANS がもつ効率的なノード管理方式 Grid および HierGrid を用いた場合に比べても，単純な LinearList を並列化した方法ながらノード数 1000 台以上の場合に優位性を確認できた．

Accerelation of JiST/SWANS wireless network simulator with GPGPU

Susumu Ishihara [†1] and Kazuki Nakashima[†2]

In simulations of wireless ad hoc networks, numerous number of packet reachability checks are needed to consider the effect of traffic of beacons which are sent periodically from all nodes. The reachability checks can be executed in parallel. Despite recent trends of general-purpose computing on graphic processing units (GPGPU), only a few cases of using GPGPU for wireless ad hoc network simulation are reported. We designed and implemented an enhancement of a Java-base wireless network simulator JiST/SWANS for using GPGPU to check the packet reachability in parallel. The experiment results show that our implementation of parallel reachability check based on the simplest node management scheme LinearList in Jist/SWANS achieves 3.4 times and 4.9 times faster than the original JiST/SWANS's LinearList when the number of nodes is 500 and 1000 respectively. It also outperforms two efficient node management schemes Grid and HierGrid in JiST/SWANS when the number of nodes is 1000 and over.

1. はじめに

都市を移動する携帯端末を持つ歩行者や交通網を行き交う自動車による無線アドホックネットワークを対象としたアプリケーションやプロトコルの開発に当たっては，実環境での実験が困難であるためにシミュレーションに依存する部分が大きい．本稿では，数千台規模の無線アドホックネットワークのシミュレーションを想定した GPUGPU（General-purpose computing on graphics processing units）による高速化について述べる．

無線アドホックネットワークのシミュレーションでは，各ノードがパケットを送信する際に，周辺のノードでの信号強度に基づいてパケットが届いたかどうかの到達判定を行っている．特に，車々間通信のように移動し続けるノードが定期的（例えば 1 秒間に 10 回）にブロードキャストでパケットを送信し続けるようなシステムの大規模なシミュレーションでは，これらの処理の頻度が高い．例えば，平均車間距離 60m の場合，片側 2 車線の道路に存在する車両の密度は両方向合わせて 64 台/km である．仮にこのような道路が 500m 間隔のメッシュ状に配置されていたとすると 5km 四方の領域には，6400 台の車両が存在することになり，これら全てのノードが送信するパケットに関しての到達性判定を行う必要がある．

信号強度の計算は，送信ノードと他のノード間の距離に基づいて行われる．この 1 対 N 間の距離計算には互いに依存性がないので，並列処理による処理速度の向上が期待できる．近年，GPGPU によるシミュレーションの高速化の試みが多く行われているが，ネットワークシミュレーションにおいては，その事例は ns2 に関して数少ない事例がある程度である[1)]．筆者らは，Java ベースのネットワークシミュレータ JiST/SWANS (Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator)[2)] におけるノード間の距離計算に伴う処理を GPGPU の統合開発環境 CUDA (Compute Unified Device Architecture) を用いて並列化した．

†1 静岡大学創造科学技術大学院
Graduate School of Science and Technology, Shizuoka University
†2 静岡大学工学部
Faculty of Engineering, Shizuoka University

Vol.2011-DPS-147 No.17
Vol.2011-MBL-58 No.17
2011/6/3

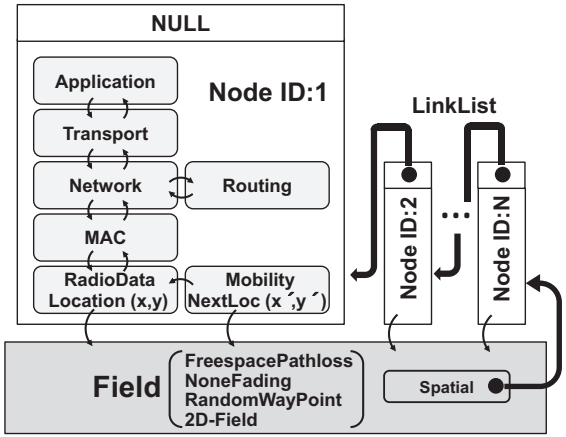


図 1 JiST/SWANS の構成
Fig. 1 Structure of JiST/SWANS

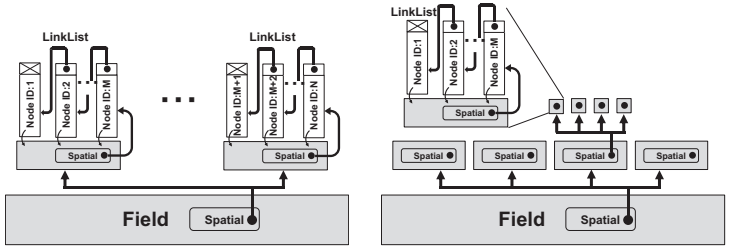
以下まず 2 章と 3 章で，本研究で用いる無線ネットワークシミュレータ JiST/SWANS と，GPGPU 環境 CUDA の概要について述べる．次に 4 章で CUDA 環境での JiST/SWANS での高速化処理の詳細について述べ，5 章でその効果の検証結果を示す．最後に 6 章に本論文のまとめを記す．

2. JiST/SWANS

2.1 JiST/SWANS の概要

JiST/SWANS[3] は，Java のみで作成された無線ネットワークシミュレータである．元々 JiST/SWANS は Cornel 大学で開発されたソフトウェアであるが，現在では同大学でのメンテナンスは行われておらず，近年では Northwestern 大学，Ulm 大学で車々間アドホックネットワークのシミュレーションをメインターゲットとしてメンテナンスされている[4)5)]．JiST/SWANS は，無線アドホックネットワークの研究分野全体で考えると，使用されている割合は ns2 などのメジャーなシミュレータには劣るが，車々間通信の分野だけでは，多くの利用事例がある．

JiST/SWANS は，JiST と SWANS の二層構成の無線ネットワークシミュレータであり，JiST 上で SWANS が動作している．JiST (Java in Simulation Time)[6] は，Java のみで作成された汎用的なシミュレーション環境である．その名前が示すように，JiST は通常の



(a) Grid (b) HierGrid

図 2 ノードのリンクリストの管理
Fig. 2 Management of node link lists

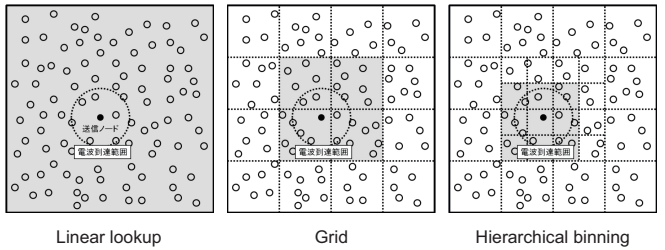


図 3 距離計算の対象エリア
Fig. 3 Target areas for calculating distance between nodes

Java のクラスでの動作タイミングをシミュレータ上での時間で動作するように，プログラム実行時に自動的にクラスのバイトコードの書き換えを行う．このため，既存の Java コード—例えば，Java で記述されたネットワークアプリケーションのコード—に少量の改変を加えるだけでシミュレーションを行うことが可能である．また，Java のガベージコレクションの仕組みをそのまま利用できるのでメモリ利用効率が良いという特徴がある．

SWANS (Scalable Wireless Ad hoc Network Simulation)[2] は JiST 上に構築された無線ネットワークシミュレーション環境である．SWANS では，**図 1** に示すように，無線ネットワークを Application，Transport，Network，Routing，MAC，Radio，Mobility，Field のそれぞれ独立したクラスの組み合わせでモデル化し，パケットの流れをシミュレートしている．後述するように，SWANS では，シミュレーション高速化のために，電波到達性の判定を効率的にできるようにノードの位置管理方法を工夫している．本研究では，GPGPU

Vol.2011-DPS-147 No.17
Vol.2011-MBL-58 No.17
2011/6/3

によりさらにこれを高速化することを目指している．

シミュレータを構成するクラスのうち，Field クラスが，ノードの移動，電波伝搬の処理に関わっている．シミュレータ実行開始時には一つの Field クラスがインスタンス化され，ノードを配置するためのフィールド（例えば 5000m × 5000m の二次元の座標空間）が用意される．このフィールドに対して Spatial クラスのインスタンスを通して，ノードが管理される．Spatial クラスはノードの管理方法を扱うクラスであり，線形リスト（LinearList），グリッド（Grid），階層型グリッド（HierGrid）の 3 つのサブクラスを持つ．最も単純な LinearList では，シミュレータ上の全ノードは一つの双方向リンクリストによって管理される．一方 Grid では，フィールドは $N \times N$ の格子で分割され，各ビンごとにノード管理用のリンクリストが用意される（**図 2**(a)）．パケットの到達可能性を調べる場合，送信元のノードの含まれているビンと通信可能距離以内の領域をカバーしているビンに含まれているノードのみを到達性判定の対象とする（**図 3**）．これにより，到達可能性判定の回数が削減されるので，シミュレーションが高速化される．HierGrid では，ノードの配置に応じて適切にビンの数を調整できるように再帰的にフィールドが 4 分割され，各ビン毎にリンクリストが用意される（図 2）．Grid，HierGrid における分割数，階層数はインスタンス生成時に変更可能である．これらの適切な値は，最小の格子の一辺の長さが最大通信距離となる値である．

3. CUDA による GPGPU

3.1 GPU

GPU (Graphics Processing Unit) とは，グラフィックス処理を行うために特化した演算装置である．GPU のアーキテクチャは一般的にコンピュータを構成する部品の一つである CPU (Central Processing Unit) とはかなり異なっている．GPU をのせたビデオカードには，GPU とビデオメモリ (VRAM) が置かれている．これらの GPU とビデオメモリは，メモリ・インターフェースで接続されており，CPU がメイン・メモリとおよそ 64bit 幅で接続されているのに対し，GPU では最大のもので 512bit 幅で接続されているため高速なデータ転送が可能である．GPU にはストリーミング・プロセッサ (SP) という演算装置が 8 基搭載されたストリーミング・マルチプロセッサ (SM) が大量に搭載されている．これらの SM ごとに同時に並列処理が可能である．NVIDIA 社の GPGPU に特化して設計されたカード Tesla C1060 には，SM が 30 基搭載されており，合計 240 基の SP が利用可能である．これらの SP が数千のスレッドを用いて同時に並列処理を行うことが可能である．

3.2 GPGPU

GPGPU (General Purpose computing on GPU) とは，GPU をグラフィックス処理以外の汎用演算に用いる技術である[7]．GPU はグラフィックス処理を行う専門の装置として開発されたが，GPU を使った演算の自由度が増すにつれて，これらの高い演算性能をグラフィックス用途以外に適応する GPGPU が生まれた．しかしながら，GPGPU は，プログラミングの際にシェーダー言語の知識や，3D グラフィクスパイプラインの知識が必要であり，グラフィックス API を使った低級言語での記述が必須であったため，簡単に使えるものではなかった．そこで，NVIDIA 社は高級言語を用いて GPGPU を行うことができる統合開発環境 CUDA (Compute Unified Device Architecture) を開発・公開している．また，AMD 社も ATI ブランドで展開する GPU 製品群向けに ATI Stream という GPGPU 環境を提供している．

CUDA は，C 言語を拡張したプログラミング言語，コンパイラ，ライブラリを含んでいる．CUDA はソフトウェア一式であり，無償で公開されている．CUDA では，Windows XP/Vista（32/64 ビット），Linux（32/64 ビット），Mac OS X がサポートされている．

CUDA のプログラム構成は，CPU を動作させるホスト・コードと GPU を動作させるデバイス・コードから成り立っている．プログラム全体の制御並びにデバイスコードの呼び出しはホストコードで記述され，GPU 内で行う処理は，デバイスコードに記述される．ホスト・コードは，通常の C 言語プログラムで記述され，デバイス・コードは C 言語を拡張させた CUDA 特有の言語で記述される．

3.2.1 CUDA プログラミングの基本フロー

CUDA の環境で GPU で計算を行うためには次の 3 つのステップ，i) 入力データのホストから GPU への転送，ii) GPU での計算, iii) 処理結果の GPU からホストへの転送，に従う．プログラムの中では，まずデバイス（GPU）側にメモリを確保する．そして，ホスト上のメモリからデバイスメモリへデータ転送を行う．次に，GPU 上で行う手続（カーネル）を呼び出して，GPU に渡したメモリ上の値を用いて GPU 上で並列演算を行う．そして，演算結果をデバイスメモリからホストメモリへ転送する．最後にデバイスメモリ内に確保したメモリ領域を解放する．

GPU 側で動作するカーネルコードは，C 言語をベースにしているが，ベクター変数や GPU 上でデータを置くメモリを指定できるといった点で C 言語とは異なる．

3.2.2 スレッド並列処理

CUDA では，GPU 内に搭載されている大量のストリーミング・プロセッサ (SP) を用い

Vol.2011-DPS-147 No.17
Vol.2011-MBL-58 No.17
2011/6/3

て並列処理を行わせている．GPU では，SP 数より圧倒的に多い数のスレッド (数千あるいは数万単位) で同時に並列処理を行うことで，高い演算能力を引き出している．CUDA では，数千を超えるスレッドをグリッドとブロックという概念を導入することで階層構造によって管理している．CUDA では，GPU 側で動作するカーネル側にグリッドと呼ばれる実行単位を設けている．グリッドの中には，ブロックと呼ばれる実行単位が (x 軸方向，y 軸方向) の二次元配置で管理されており，ブロックの中には CUDA における演算処理の最小単位であるスレッドが (x 軸方向，y 軸方向，z 軸方向) の三次元配置で管理されている．CUDA では，グリッド内のブロック数およびブロック内のスレッド数を自由に指定することができる．CUDA の仕様では，SP に対して最高で 65535 × 65535 × 512 個のスレッドの実行を行うことが可能である．ブロック内のスレッド数には上限があり，512 個までとされている．

GPU は SIMD 型であるため，各 SM に配置された 8 個の SP が同時に並列処理を行っている．このとき，GPU では，32 個のスレッドが Warp と呼ばれる単位で管理されている．そして，32 個のスレッド毎に同じ命令を実行している．したがって，ブロック内のスレッド数を 32 の倍数にしたとき最も効率が良い．

3.2.3 メモリ・モデル

CUDA では，GPU のメモリを使いやすくするために，ハードウェア名とは異なる独自のメモリ・モデルが導入されている．CUDA では，GPU の内部に多くの種類のメモリが階層的に配置されている．デバイス上の複数のメモリは，ビデオカード上に搭載されているメモリ (オフチップ・メモリ) と GPU 内に搭載されているメモリ (オンチップ・メモリ) に分けられる．オンチップ・メモリは，容量は小さいが，非常に高速なアクセスが可能となっている．オンチップ・メモリの中でも，シェアード・メモリは，プログラムの中から直接値を変更することもできるため，便利である．CUDA では，GPU 上で並列処理を行う際に，どのメモリ上で行うか指定することができるため，高速化を行うにはオンチップ・メモリをうまく使えるよう工夫することが重要である．

4. GPGPU による JiST/SWANS でのパケット到達判定処理の並列化

JiST/SWANS は，Cornel 大で開発されたシミュレータであるが，現在では Ulm 大の研究グループによって高速化のために拡張されている．本研究では Ulm 大学で拡張された JiST/SWANS[5] をベースに GPGPU による拡張を行った．Ulm 大の拡張では，位置管理に関するいくつかのバグフィックスが行われている他，Java の新しいライブラリへの対応処理が行われている．

2 章で述べたように JiST/SWANS は Java のみで作成されているため，C 言語をベースとした CUDA を直接利用することができない．そこで，CUDA を Java に対応させた jCuda (Java bindings for CUDA)[8] を用いて JiST/SWANS を GPGPU に対応させるようにした．jCuda は，別途作成した GPU バイナリ（PTX）を呼び出して利用するためのラッパーと，CUDA が提供する各種ライブラリ（CUBLAS，CUFFT など）を呼び出すための API によって構成されている．

4.1 GPGPU による並列化の戦略

JiST/SWANS でのパケット到達判定は，以下の手順で行われている．まず，送信ノードと判定対象の相手ノードの位置座標に基づいて 2 ノード間の距離を計算し，その距離に基づいてパスロスを求める．さらにフェージングによる減衰を求め，これらと送受信側のアンテナゲインに基づいて，受信強度を求め，最小受信強度と比較することで信号到達の成否を判定する．これらの処理のうち，最小受信強度との比較部分以外を GPU 内部で実行する手続き（カーネル）として実装した．

JiST/SWANS で実際にパケット到達判定処理が行われているのは，Field（jist.swans.field.Field）クラスである．このクラス内での処理で，jCuda の機能を介してパケット到達判定処理を行う GPU カーネルを呼び出すようにした．また Field クラスのコンストラクタで，GPU-ホスト間でのデータ交換用の配列を格納するメモリ領域の確保，カーネルの GPU への転送処理の他，GPU 内でのフェージングの計算のための乱数生成器の初期化を行うカーネルの呼び出しを行うようにした．

今回の実装・実験では，対地反射 2 波モデルによるパスロスとレイリーフェージングを想定した．なお，JiST/SWANS では，用意されたパスロスとフェージングモデルのクラスを自由に選択可能であるが，今回の実装では，特定のパスロスとフェージングモデルを組み合わせて受信電力強度を計算する処理を一つの関数としてまとめてカーネルに実装した．カーネルでは送信元のノードとフィールド内の全ノード N 台との間の到達判定処理を N 本のスレッドで並列計算する．

4.2 GPU—ホスト間のデータ転送

パケット到達判定処理で必要なノード間の距離を得るためには，各ノードの現在の位置座標を GPU に渡す必要がある．CUDA では，通常の C 言語におけるメモリ間データ転送のプログラミングと同様に，ホスト-GPU 間のデータ転送を行うために，転送元と転送先の先頭アドレスと転送サイズを指定する．つまり，転送されるデータは連続した領域に配置さ

Vol.2011-DPS-147 No.17
Vol.2011-MBL-58 No.17
2011/6/3

れていることが前提である．しかし，JiST/SWANS 内部でのノード管理方式では，複数のノードの位置座標が連続領域に格納されているわけではない．従って，ホストから GPU への位置座標の転送に当たっては工夫が必要である．

2 章で述べたように，JiST/SWANS では，ノードの無線関連情報をまとめたクラス (RadioData) のインスタンスがリンクリストで管理されている．また，このインスタンスは同時に 1 次元配列でも管理されている．具体的には，1 次元配列には，いちど生成された全ての RadioData クラスのインスタンスが，削除されたものも含めて納められており，リンクリストには，削除されていないインスタンスのみが納められている．RadioData クラスには，ノードの位置情報を含むクラス，移動モデルを表すクラス，リンクリストにおける接続関係などが格納されている．これら RadioData クラスのメンバ変数は他のクラスへの参照であるので，RadioData クラスの配列があるものの，ノードの位置座標を表す数値データが連続したメモリ領域上に配置されているわけではない．

そこで，GPU に転送するべき全ノードの位置座標，ゲインを連続領域に配置しておくため，これらの値を格納する別の配列をそれぞれ用意しておき，ノードの移動時には，RadioData クラスの値と連動してこれらの配列の値も更新されるようにした．GPU での到達性判定処理を行うときには，上記の各座標とゲインの配列内の数値を全て GPU に転送する．ただし，前回到達性判定処理が行われてからノードが移動していない場合は，転送処理を行わない．これにより，GPU 使用時のオーバーヘッドとなるホスト-GPU 間のデータ転送処理の回数を削減している．なお，ここで「移動していない」かの判定基準は，JiST/SWANS 内での位置更新精度に依存する．JiST/SWANS のランダムウェイポイントモデルでの位置更新は，決められた距離以上の移動を行わない限り行われない．本稿での評価では，3m 以上の移動した場合「移動した」と見なすこととした．

5. 性能評価

5.1 検証シナリオ

前章で述べた方法をノード管理アルゴリズム LinearList の場合に対して実装し，**表 1** に示すハードウェアを搭載したマシンを用いて動作検証を行った．OS には，Ubuntu 10.04（Linux カーネル 2.6.32-24）を利用し，CUDA 3.2 および jCuda 0.3.2 を利用した．5000m × 5000m の二次元平面のフィールドを用意し，その上をノードがランダムウェイポイント移動モデル（ポーズタイム: 0 秒，最低速度: 0m/s，最高速度: 15m/s）に従って移動する様にした．各ノードは 0.1 秒毎に UDP パケットをブロードキャストする．パケットのペイロードのサイズを 12bytes とした．このパケットを受信したノードは，転送を行わず，受信処理のみを行う．対地反射 2 波モデルによるパスロスとレイリーフェージングを想定した．MAC 層のプロトコルには IEEE802.11b を用いた．パケットの到達距離は距離減衰やフェージングを考慮したうえで，平均でおよそ 370m である．GPU を用いた場合の Linear lookup と用いていない場合の LinearList，Grid (フィールドを 13 × 13 に分割)，HierGrid (階層数 5) において，上記のシナリオを用いて，シミュレーション上の時間で 60 秒間のシミュレーションを行い，シミュレーション実行時間の平均値を求めた．Grid，HierGrid における分割数，階層数はパケット到達距離から算出した値であり，予備実験により最適であることが確認されている．なお，GPU の各ブロック内のスレッド数は 128 とした．この値は GPU カーネルで使用するレジスタ数，共有メモリの容量をもとに NVIDIA 提供の CUDA Occupancy Calculator で算出した最適値である．

表 1 ハードウェアの仕様
Table 1 Hardware specification

	CPU	GPU	
Processor	Intel Core i7	Tesla C1060	GeForce GTS 250
SP 数	—	240	128
SP 動作クロック	—	1296MHz	1836MHz
メモリ転送速度	—	102GB/s	70.4GB/s
メモリ容量合計	3 GB	4GB	512MB

5.2 測定結果と考察

図 4 および **表 2** に，フィールド内に存在するノード数を変化させたときのシミュレーション実行時間を示す．シミュレーションは，ノード数 1000 台未満の場合はそれぞれ 10 回，1000〜2000 台の場合 5 回，4000 台の場合に 1 回行った．ただし低速な LinearList ノード管理方式については，800 台未満の場合に 10 回，それ以上では 1 回ずつのみとした．

LinearList での実行時間はノード数の 2 乗にほぼ比例している．GPU を用いた場合は，ノード数が少ない（50 台）ときには，GPU 使用に伴うオーバーヘッド（GPU へのカーネル転送，データ転送，メモリ確保）のために実行時間が LinerList に対して大きくなっているが，ノード数が 100 台からノード数 1000 台未満の場合は実行時間の増加率はノード数の 1〜1.2 乗程度にとどまり，ノード数 200 台以上では LinerList よりも高速にシミュレーションが実行できている．GPU 使用による速度向上比は，ノード数 500 台の場合で約 3.4 倍，1000 台の場合で約 4.9 倍である．GPU が搭載する SP 数，240 および 128 に対して，この

Vol.2011-DPS-147 No.17
Vol.2011-MBL-58 No.17
2011/6/3

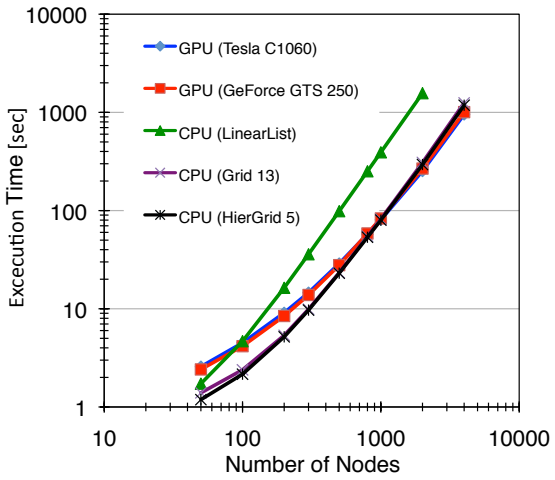


図 4 シミュレーション実行時間の比較
Fig. 4 Simulation execution time

速度向上比は小さいが，これは本シミュレーションで行っている GPU 上の処理量が大きくはない一方で，ノードの位置座標と処理結果転送のホスト-GPU 間の転送に起因するオーバーヘッドが大きいためである．今回の実装では，位置座標は GPU 上のグローバルメモリに格納している上，フィールド内の全ノードでの受信信号強度を処理結果として毎回 GPU からホストに転送しているので，このオーバーヘッドの影響が大きい．

Grid および HierGrid を用いると，LinerList の場合に比べて大幅な実行時間短縮が実現できており，ノード数が 1000 台以下の場合には GPU を用いた場合の同等以上の速度を達成している．これは，フィールドを分割して，通信可能な範囲のノードを対象としてパケット到達性の判定を行ったことによる．ところが，ノード台数が大きくなると，Grid および HierGrid におけるビンの中に含まれるノード数が大きくなるために，GPU を用いた場合に比べて優位性が薄れる．つまり，ノード数が 1000 を超えた場合，単純に LinearList を GPGPU 対応にした場合でも，GridHierGrid よりも高速にシミュレーションが実行できる．

GPU の動作クロックと SP 数の違いは性能の差に表れた．ノード数が少ない場合，よりクロックの速い GeForce GTS250 の方が Tesla C1060 よりも若干高速にシミュレーションが実行できた．一方で，ノード数が大きい場合，クロックが遅いものの SP 数が多く，より多くのスレッドの同実行が可能な Tesla C1060 の方が高速にシミュレーションが可能であった．

表 2 シミュレーション実行時間 [秒]
Table 2 Simulation Executing Time [sec]

Number of nodes	GPU		CPU		
	Tesla C1060	GeForce GTS 250	LenearList	Grid (13)	HierGrid (5)
50	2.574	2.411	1.727	1.393	1.188
100	4.498	4.160	4.676	2.389	2.157
200	9.099	8.443	16.351	5.363	5.179
300	14.585	13.819	35.907	9.892	9.694
500	28.896	27.819	98.617	23.279	23.049
800	57.573	58.481	251.405	54.720	53.537
1,000	80.773	83.125	392.176	81.692	79.940
2,000	252.765	267.886	1568.371	308.263	294.301
4,000	953.217	1,006.477	—	1,249.032	1,182.436

5.3 検　　討

以上の結果を踏まえた上で，GPGPU による並列処理を用いてさらなる高速化を行うための戦略について検討する．

5.3.1 Grid と HierGrid における並列化

LinearList に比べて計算コストを削減するように工夫されたノード管理アルゴリズム Grid を用いた場合における GPU による並列化の方法を考える．Grid を用いれば，処理結果の転送量の大幅削減が期待できるが，その一方で，位置座標データの複雑な管理に伴う処理量の増加に注意する必要がある．Grid では，等分割された区画毎にリンクリストが存在するため，GPU へ転送用の配列も区画毎に必要になる．さらに，LinearList では一つのリンクリストで管理されているノードのオブジェクト数が固定であったのに対し，Grid ではノードのビン間の移動に伴って，リンクリストへのノードの追加と削除が頻繁に行われる．また，パケット到達性の判定は複数のビンに含まれるノードに対して行われるので，GPU に位置位置座標を転送するための配列の内容は，位置判定の都度更新される必要がある．従って，この配列の管理のオーバーヘッドをできるだけ少なくするような工夫をしないと，Grid ならびに GPU を用いたことによる高速化の効果を得ることが困難だと考える．同様の議論は，位置管理アルゴリズム HierGrid の場合でもあてはまる．

5.3.2 GPU 内でのノードの位置情報の管理

GPU へのデータ転送に伴うオーバーヘッドの削減のために，全ノードの位置情報を常に

Vol.2011-DPS-147 No.17
Vol.2011-MBL-58 No.17
2011/6/3

GPU 内に置いておくことを考える．しかし，各ノードは移動しているため，位置情報を更新しなければならない．そこで，ノードが移動するたびに，移動したノードのみの位置情報を GPU に転送して位置情報を更新する，あるいは，パケット到達性の計算をするときに，それまで更新していなかった GPU 上のノードの位置情報をまとめて更新するようにする．この方法を用いた場合，コピー対象となるデータのメモリ上の位置が連続しているとは限らないので，コピー対象のデータの個数が多い場合には，オーバーヘッドが大きくなると考えられる．一方，通信頻度が小さい場合には，位置情報の更新処理が小さくなるので，この方法の効果は大きくなると考える．

6. ま と め

Java ベースの無線ネットワークシミュレータ JiST/SWANS におけるパケット到達性判定処理を GPGPU により並列処理する方法を設計・実装し，実験によりその効果を確かめた．JiST/SWANS がもつ最も単純なノード管理方式 LinearList にたいして GPGPU による並列化を導入した結果，5000m 四方の領域での IEEE802.11b での通信を想定したシミュレーションにおいて，ノード数 500 台の場合で約 3.4 倍，ノード数 1000 台の場合に約 4.9 の速度向上を確認できた．また，JiST/SWANS がもつ効率的なノード管理方式 Grid および HierGrid を用いた場合に比べても，単純な LinearList を並列化した方法ながらノード数 1000 台以上の場合に優位性を確認できた．今後，Grid，HierGrid を用いた場合にも GPGPU による並列化を試みる他，ボトルネックとなっているホスト-GPU 間のメモリ転送処理に工夫を行い，さらなる高速化を行う予定である．

謝辞 本研究は，科学研究費補助金挑戦的萌芽研究 「自律移動困難な移動センサネットのための通信スケジューリング方式の開発（課題番号 22650011）」の助成によるものである．ここに記して謝意を示す．

参 考 文 献

1) Godjoska, T., Filiposka, S. and Trajanov, D.: Parallel Execution of the NS-2 Sequential Simulator on a GPU, *Proc. 19th Telecommunications Forum (TELFOR 2011)*, pp.159–162 (2010).
2) Barr, R.: *SWANS - Scalable Wireless Ad hoc Network Simulator User Guide*, http://jist.ece.cornell.edu/docs/040319-swans-user.pdf (2004).
3) *JiST - Java in Simulation Time / SWANS - Scalable Wireless Ad hoc Network Simulator*, http://jist.ece.cornell.edu/.
4) Choffnes, David R and Bustamante, Fabian E and Northwestern Univ Evanston il Dept of Computer Science: Straw-an integrated mobility and traffic model for vanets, *International Command and Control Research and Technology Symposium (CCRTS)*, No.10 (2005).
5) *Extensions by Ulm University — vanet.info*, http://vanet.info/node/12.
6) Barr, R.: *JiST - Java in Simulation Time User Guide*, http://jist.ece.cornell.edu/docs/040319-jist-user.pdf (2003).
7) Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. and Purcell, T.: A Survey of General-Purpose Computation on Graphics Hardware, *Computer graphics forum*, Vol.26, No.1, pp.80–113 (2007).
8) *jcuda.org - Java bindings for CUDA*, http://www.jcuda.org/.