Vol.2019-ARC-240 No.19
Vol.2019-SLDM-191 No.19
Vol.2019-EMB-53 No.19
2019/2/28

経路探索処理向け専用ハードウェアの検討

江崎 ゆり子[1]　坂本 龍一[2]　近藤 正章[2,3]

概要：最短経路探索問題とは，重み付きグラフにおいて 2 つのノードを結ぶコストが最小となる経路を求める問題である．最短経路探索問題を解く A*アルゴリズムは経路ナビゲーションシステムやロボットの自動計画，VLSI 設計など，様々な分野に応用されている．しかしながら，情報量の増大とともにグラフ規模は大規模なものとなることが予想され，より高速かつ省電力に問題を解くことが重要である．本研究では最短経路探索問題について A*アルゴリズムを用いて解く専用ハードウェアを FPGA を用いて HDA* (Hash Distributed A*) をベースに作成し，実行時間と電力について評価を行った．さらに，マルチコア化やコア内部の並列化を行い，実行時間のシミュレーション結果を比較した．

キーワード：最短経路問題，A*アルゴリズム，ソーティングネットワーク，FPGA，高位合成

1. はじめに

グラフ理論における経路探索問題とは，重み付きグラフの与えられた 2 つのノード間を結ぶ経路を求める問題である．一般に経路の重みの総和，すなわちコストが小さいほどよい解となる．経路探索問題の解法は，Google マップのルート検索や NAVITIME の乗り換え案内などの経路ナビゲーションシステムに利用されている．また，VLSI (Very large scale integration) のゲート・配線の配置問題 [1] やアミノ酸の配列問題 [2] など様々な分野でも応用されている．この問題における探索は，グラフについて，ノードを状態，エッジを状態遷移とみなすと，グラフは状態空間探索となる．そのため，自動運転技術における経路計画 [3] や自律移動ロボットの動作計画 [4] などへも応用されている．

エッジの重みが非負の場合の単一始点経路探索問題の代表的な解法として，ダイクストラ法 [5] と A*アルゴリズム [6] が挙げられる．いずれも最良優先探索を行うアルゴリズムであり，最適解を発見することを保証する．コストの下限値が既知の場合，A*アルゴリズムでは，あるノード n を通る経路の最小コストの推定値を評価関数 $f(n)$ として用いることで，ダイクストラ法よりも高速に問題を解くことが可能である．さらに，PRA*[7] や HDA*[8] と呼ばれる並列化による高速化手法が提案されている．これらの並列化によってより大規模な問題を高速に解くことができるようになっている．

経路探索は様々なグラフ問題に用いられるが，実世界で現れるソーシャルネットワークグラフや，自動運転技術に用いられる地図グラフのノード数は膨大なものとなる．さらに，今後の応用範囲の拡大とともに，ますます情報量は増大し，グラフ規模は大規模なものとなることが予想されるため，さらなる経路探索の高速化が重要である．また，計算に要するエネルギーを削減することも重要である．特に，自動運転車や自律移動ロボットにおいては，バッテリー駆動のため省エネルギー化がより重要である．そのため，本研究では高速に最短経路問題を解く専用回路設計と性能分析を示す．

本研究では，経路探索アルゴリズムの一般的な並列化手法である HDA*をベースに，経路探索を専用ハードウェア化する方法について示す．本研究の貢献は下記の 3 つである．

- 専用ハードウェア設計：
  既存の HDA*アルゴリズムに対し，ハードウェア化すべき際のポイントについて明らかにし，高速化を実現するためのハードウェア構成を示す．
- GPU・CPU との比較：
  提案する専用ハードウェアと GPU・CPU の実行時間を比較し，経路探索専用ハードウェアの有用性について示す．
- 高速化のためのチューニングと分析：
  コアの並列化，ループの回路展開による高速化を行い，

[1] 東京大学工学部計数工学科
[2] 東京大学大学院情報理工学系研究科
[3] 理化学研究所 R-CCS

© 2020 Information Processing Society of Japan

リソース量について示す．

続く第 2 章では関連研究を示し，3 章では経路探索アルゴリズムの詳細について述べる．4 章では専用ハードウェアの設計と実装について述べる．5 章に評価を示し，6 章ではまとめについて示す．

2. 従来手法とその問題点

2.1 経路探索問題

最短経路探索問題とは，ノードの集合 V，ノード間の連結関係を表すエッジの集合 E，エッジのコストの集合 C からなる重み付き無向グラフ $G = (V, E, C)$，スタートノード $s \in V$，ゴールノード $t \in V$ が与えられたとき，s と t を結ぶ経路の中で，エッジの重みの総和が最小となる経路を求める問題である．

この最短経路問題についての解法は，エッジの重みに負値が存在するかで異なる．エッジの重みに負値が存在する有向グラフにおいて，負閉路 (エッジの重みの総和が負となる閉路) が存在しない場合，ベルマン-フォード法 [9] で解くことができる．負閉路が存在し，最短経路が定まらない場合でも，ベルマン-フォード法では負閉路の検出が可能である．また，エッジの重みが全て非負値であるグラフにおいて，最短経路における距離について，部分構造最適性が成立している．すなわち，グラフにおける s から t までの最短経路 $P_{s,t}$ とし，$u, v \in V$ が最短経路上に存在する 2 ノードとすると，$P_{s,t}$ は $P_{u,v}$ を必ず含む．この最適性から，動的計画法の一種であるダイクストラ法を用いて効率よく問題を解くことができる．さらに，ダイクストラ法に経路の見積もりを用いて探索するというヒューリスティックな手法を併用した “A*アルゴリズム” を用いると，ダイクストラ法よりも高速に解くことができる．

2.2 A*アルゴリズムの概要

一般的には A*アルゴリズムは 2 つのリストを用いて実装される．2 つのリストを用いて次に選択できるノードの候補と一度選択したノードを記憶し，同じノードを二度選択しないために利用する．この実装では，グラフの規模が小さい場合は現実的な時間で解くことが可能である．

しかし，グラフの規模が大きい場合は，一度選択した全てのノードを記録する必要があるため，リストに入るノード数は指数的に増加していく．A*アルゴリズムを実行する際は，状態空間であるグラフの規模が大きくなるにつれ，探索済みのノードを記憶するためメモリが必要となる．探索済みのノードは指数的に増大していくため，メモリ消費も指数的に増大し，膨大な量のメモリが必要となる．また，探索空間が大きくなるにつれ，問題を解くために必要な計算時間も増えていく．

2.3 A*アルゴリズムの高速化

上記の計算時間増大への対処として．A*アルゴリズムの実装ついて様々な手法が提案されている．例えば，探索済みのノードを記憶せず，今まで探索された最大の f 値のみ保存し，探索空間を次第に大きくしていく IDA* (Iterative Deepening A*) [10] などである．しかし，同じノードを何度も選択し展開するため，特にヒューリスティック関数 $h(n)$ が重い場合は計算に時間がかかる．そこで，A*アルゴリズムをマルチコアプロセッサを用いて単純な並列化を行った PRA* (Parallel Retraction A*) [7] が提案されている．同じく A*アルゴリズムの並列化手法で，プロセス間のノードの送受信をハッシュ関数を用いて非同期的に行う HDA* (Hash Distributed A*) [8] も提案されている．ハッシュ関数はロードバランスの良い Zobrish Hash[11] を用いているため，効率よくノードを分配できる，そして，ノードの送受信をハッシュで行うことで，あるノードについて送られるプロセスはただ一つとなるので，各プロセス間にリストを保持することができるため，メモリ環境が分散化できることがこの手法の一番の利点である．さらにノードの送受信方法について，探索空間を抽象化して隣接するノードは近隣のプロセスに送る手法 [12] も提案された．

さらに，IDA*や PRA*，HDA*などはマルチコア CPU (Central Processing Unit) による実装に焦点を当てていたが，GPGPU (General-Purpose computing on Graphics Processing Units) を用いて大規模な A*アルゴリズムの並列化を行う手法 [13] も提案された．

本研究では，経路探索問題を解く専用ハードウェアを考案し，そのハードウェアを用いて探索の高速化を実現することを目的としている．グラフの巨大化はハードウェアの進歩を上回っており，高速に経路探索問題を解くためのハードウェア設計が必要である．そこで，ハードウェアとして FPGA (Field-Programmable Gate Array) を用いることで自由度の高い設計を可能とするための実装法を提案する．なお，本研究では最短経路問題を解くハードウェアを検討するが，対象はグラフのエッジの重みは全て非負値であり，またスタートノードとゴールノードの数は両方とも 1 つである問題に限定する．そこで，問題を解くのに利用するアルゴリズムは前述の理由から A*アルゴリズムとする．

3. 経路探索アルゴリズム

3.1 A*アルゴリズムの詳細

A*アルゴリズムでは，あるノード n における評価関数

$$f(n) = g(n) + h(n) \tag{1}$$

をもとに探索を行う．ここで，$g(n)$ はスタートノード s から n までのコスト，$h(n)$ は n からゴールノード t までの推定コストであり，ヒューリスティック関数と呼ばれる．評

価関数 $f(n)$ は，ノード n を探索時点での s から n を通り，t までたどる経路の最小コストの推定値であるといえる．

始めにスタートノード s に隣接するノードの f 値を計算する．s は探索済みとし，f 値を計算した隣接するノードは探索待ち状態とし，情報を保持しておく．この一通りの手続きを本稿では今後，「展開」と呼ぶ．探索待ち状態のノードの中から，最も f 値の小さいノードを選び，そのノードを展開する．これを繰り返し，ゴールノード t が選ばれたら終了とする．また，t が選ばれる前に，探索待ち状態のノードがなくなった場合は，s から t での経路が存在しないとして終了する．A*アルゴリズムの疑似コードをアルゴリズム 1 に示す．

Algorithm 1: A* search

```
Initialization: OPEN = {s} with f(s) = h(s);
while OPEN ≠ ∅ do
    Get and remove from OPEN the node n with the
      lowest f(n);
    if n == t then
        Return solution path from s to n;
    for each successor n' of n do
        g' = g(n) + cost(n, n');
        if n' ∈ CLOSED then
            if g' < g(n') then
                Move n' from CLOSED to OPEN;
            else
                continue;
        else
            if n' ∈ OPEN then
                Add n' to OPEN;
            else if g' ≥ g(n) then
                continue;
        Set g(n') = g';
        Set f(n') = g(n') + h(n');
        Set parent(n') = n;
    Add n to CLOSED;
```

$C^*(n)$ をノード n からゴールノード t までの探索時点での最小コストとすると，ヒューリスティック関数 $h(n)$ が

$$h(n) \leq C^*(n) \tag{2}$$

を満たすとき，$h(n)$ は許容的 (admissible) という．すなわち，$h(n)$ は，$C^*(n)$ の下限となる．

また，$c(n, n')$ をノード n とノード n' を結ぶ辺のコストとすると，ヒューリスティック関数が

$$h(n') = h(n) + c(n, n') \tag{3}$$

を満たすとき，$h(n)$ は単調 (consistent または monotonic) という．

$h(n)$ が許容的なヒューリスティック関数のとき，$h(n)$ を用いた A*アルゴリズムは，大域的最適解を返す．$h(n)$ が単調ならば，許容的となるため A*アルゴリズムは大域的最適解が求まる．しかし，$h(n)$ が単調でない場合でも許容的であれば，大域的最適解を返す．

3.2 A* アルゴリズムの実装

A*アルゴリズムでは，前節の疑似コードにあるように，探索中のノードを格納しておく $OPEN$ リストと探索済みのノードを格納しておく $CLOSED$ リストを用意する．$OPEN$ リストを実現するデータ構造に必要な要素は以下の 2 つである．

- 最小の f 値をもつノードが参照でき，かつ取り出せること
- 新たにノードが挿入できること

CPU におけるソフトウェアの実装などでは，一般的に $OPEN$ リストは優先度付きキュー (プライオリティーキュー) や赤黒木などの二分探索木で実装される．例えば，優先度付きキューは，ヒープで実装した場合，リスト内のノードの数を N とすると，先頭のノードの参照の計算量は $O(1)$，取り出しの計算量は $O(\log N)$，挿入の計算量は $O(\log N)$ である．ヒープは，葉ノード (末端のノード) を除いたノードは必ず 2 つの子ノードをもち，葉は左詰であり，親ノードは必ず子ノードよりも大きい値をもつ木構造である．ヒープで実装された優先度付きキューは f 値が最小のノードを取り出すときは親ノードを取り出せばよく，新たなノードの挿入もできるため，上の $OPEN$ リストに必要な要素を満たすことができる．一方で，A*アルゴリズムの実行時間の大部分は $OPEN$ リストに関する操作 (挿入・取り出し) に費やされており，この部分の高速化が重要である．

3.3 HDA* アルゴリズムの実装

A*アルゴリズムを並列化し高速化する手法は多く提案されてきたが，大きく分けて異なる点は，

- メモリを共有してもつか分散化するか．
- 各プロセス間の通信は同期か非同期か．
- ノードの分配方法

である．

本研究では HDA*をベースにする．すなわち，メモリを分散してもち，各プロセス間の通信は非同期で行う．また，ノードはハッシュ関数を用いて分配する．Algorithm 2 に HDA*アルゴリズムの疑似コードを示す．プロセスごとに $BUFFER$ を持ち，$OPEN$ リストを分散して持つ．擬似コード中の下から 4 行目では Hash 計算によって次に計算を行うプロセス番号を決定し，対象のプロセス $BUFFER$ にノード情報を送信する．本通信は計算とは非同期に行うことで性能向上を行うことができるが，計算時間が比較的短いため，本非同期通信がボトルネックとなりやすいと考

えられる．

Algorithm 2: Hash Distributed A*

```
Initialization: incumbent.cost = ∞ ;
Initialization: OPEN_{Hash(s)} = {s} with f(s) = h(s);
while TerminateDetection() do
    while BUFFER_p ≠ ∅ do
        Get and remove from BUFFER_p the node n'
          with (n, g');
        if n' ∈ CLOSED_p then
            if g' < g(n') then
                Move n' from CLOSED_p to OPEN_p;
            else
                continue;
        else
            if n' ∈ OPEN_p then
                Add n' to OPEN_p;
            else if g' ≥ g(n') then
                continue;
        Set g(n') = g';
        Set f(n') = g(n') + h(n');
        Set parent(n') = n;
    if OPEN_p ≠ ∅ then
        Get and remove from OPEN_p the node n with
          the lowest  f(n);
        if f(n) ≥ incumbent.cost then
            continue ;
        Add n to CLOSED_p;
        if n == t then
            if f(n) < incumbent.cost then
                incumbent = path from s to n ;
                incumbent.cost = f(n);
        for each successor n' of n do
            Set g' = g(n) + cost(n, n');
            Add n' with (n, g') to BUFFER_{Hash(n')};
if incumbent.cost = ∞ then
    Return failure (no path exists) ;
else
    Return solution path from s to n ;
```

4. 専用ハードウェアの実装

本章では初めに HDA*の専用ハードウェアを設計する際のアプローチについて述べ，設計する専用ハードウェアの全体構成，各モジュールについて示す．また，実装方法についても説明する．

4.1 HDA*のハードウェア化へのアプローチ

これまで述べたように HDA*アルゴリズムでは

- ソートが頻繁に行われ，ソートに要する時間が多い
- コア間で非同期に小サイズなデータを頻繁にやり取りする必要があり，通信オーバヘッドが大きい

ことが問題であると考えられる．そこで，これらの問題をハードウェア化することによって解決し高速化を図る．具体的には，

- ソーティングネットワーク回路によるソートの高速化
- 専用パケットネットワークによる非同期通信の高速化

を行う．さらに，周囲ノードの探索やヒューリスティック関数，ハッシュ計算もすべてハードウェア化することにより高速化を図る．また，探索やヒューリスティクス関数部のループ展開を行うことによる高速化についても検討する．

4.2 基本的なハードウェアの構成

前述の仕様を実現するために，ハードウェアの基本構成を図 1 として作成した．この基本構成を，$\mathrm{FPGA}_{1,1}$ と呼ぶ．A*アルゴリズムを実行するプロセスコアと各種データを格納する Global データテーブルから構成される．また，1 プロセスの場合を示している．

まず，プロセスには展開を行いたいノード番号と f 値をセットにしたデータが innode_stream に到着する．stream とあるようにこれらのデータは FIFO に格納される．FIFO に溜まったデータはハードウェアによる専用ソート回路を用いて高速にソートされる．この際，f 値が昇順になるようにソートを行う．入力である $OPEN$ リストは sorted_list_a[] に格納され，ソート回路を経由してソート済みデータが sorted_list_b[] に格納される．新しいノードが入ってくるごとに，ソーティングネットワークを用いて 2 つのリスト間でソートを行い，ソートの方向は 1 回ソートを行うごとに変更することで，配列値のコピーを省くことができる．これによって新しいノードデータの到着と expanding から computing までの手続きを並列に動作させることができる．

extracting ではソート済みのリストから f 値が最も小さいノード情報を取り出す作業を行う．取り出したノード情報をもとに expand ではノードの展開を行う．この際，Global データ中にある graph_table に格納されているノード情報を参照し，expand するかを決定する．

4.3 ソーティングネットワーク

ソーティングネットワーク [14] は，ワイヤ・コンパレータから構成される，数列のソートを行う数理モデルのことである．ワイヤは値を伝搬し，コンパレータは入出力に 2 本のワイヤをとり，入力の値の大小に対して出力が決定される．

ソーティングネットワークは，コンパレータの接続方法によって，シェルソートやバイトニックソート，バッチャー奇偶マージソートなど様々なネットワークが提案されてきた．上記で例として述べた 3 つのソーティングネットワークは，いずれのも要素数 n に対して，大きさ $O(n(\log n)^2)$ かつ深さ $O((\log n)^2)$ のソーティングネットワークである．

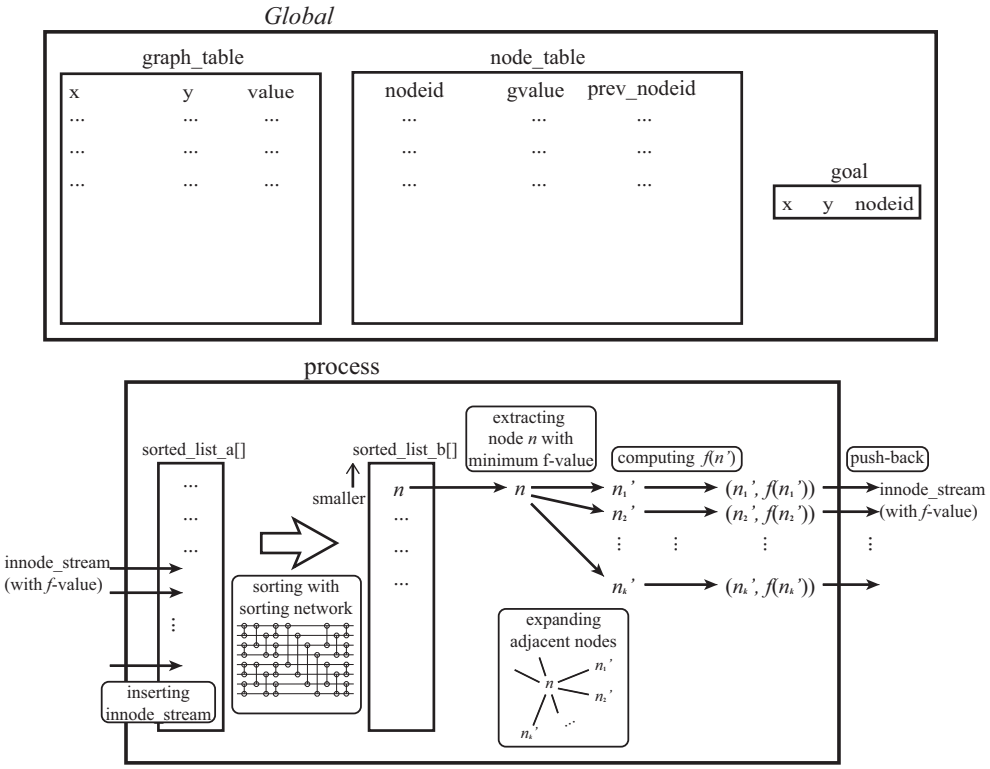

図 1 専用ハードウェアの構成

FPGA 上ではソーティングネットワークを用いて，ソートを行う手法 [15] が多く提案されている．また，ソーティングネットワークとマージソートツリーソートを組み合わせることで，FPGA を用いて面積性能効率の良いソートのアクセラレータを実現する手法 [16] も提案されている．

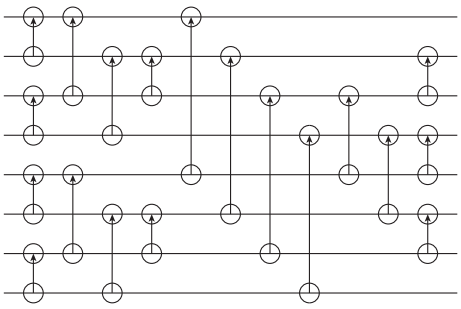

図 2 バッチャー奇偶マージソート ($n = 8$)

本研究では，新しいノードが挿入されるたびにソーティングネットワークを用いてリストのソートを行うことで，*OPEN* リストを実装した．ソーティングネットワークはバッチャー奇偶マージソートを用いた．バッチャー奇偶マージソートを用いた理由としては，構成が再帰的でありハードウェアに使用しやすいためと，表 1 からわかるように，コンパレータの個数が他の 2 つのソーティングネットワークと比較して少なく，電力性能という観点で優れているためである．

表 1 各ソーティングネットワークにおける要素数 n に対するコンパレータの個数

	number of comparator		
n	shellsort	bitonic sort	odd-even mergesort
4	6	6	5
16	83	80	63
64	724	672	543
256	5106	4608	3839
1024	31915	28160	24063

4.4 HDA*をベースにしたマルチコア化

図 1 では A*をベースとした 1 プロセス構成の設計を示した．これを，HDA*をベースにプロセスコアの並列化を行う．図 3 に並列化したプロセスコア部の詳細を示す．各プロセスコアの構成は A*の場合とほぼ同様であるが，展開したノード情報を各プロセスコアに分配するための仕組みを追加した．具体的には，プロセスコア内で展開したノード番号を基にハッシュ関数を実行し，展開したノードを次に計算するプロセスコア番号を計算する．図 1 において，プロセスコア数が i であるハードウェアの構成を $\mathrm{FPGA}_{i,1}$ と呼ぶ．push-back ではプロセス番号を基に，次のノードの計算を行うプロセスコアに計算を依頼する．また，これらの依頼はクロスバースイッチを通じて他のプロセスコアの innode_stream に入力される．送信するデータ量は小さく非同期に通信が発生するため，ソフトウェア実装ではオーバヘッドが大きいと考えられる．そのため，ハッシュ計算回路，クロスバースイッチを回路として実装すること

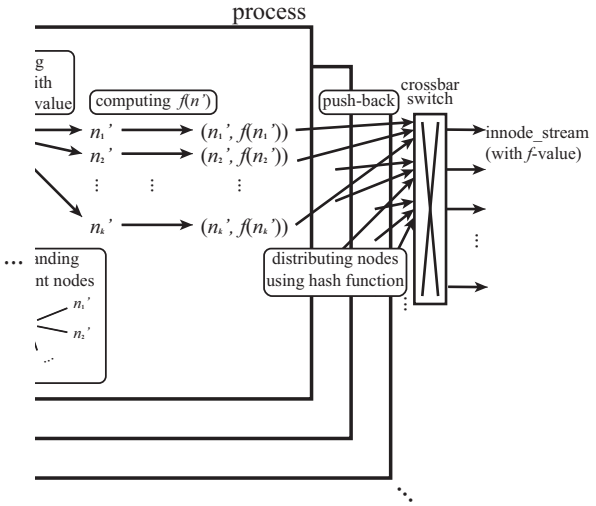

図 3　HDA*をベースにしたマルチコア化

で通信オーバヘッドの削減が期待できる．

4.5　コア内部の並列化

前節ではプロセスコアを並列化することによる高速化方法について示したが，本節ではプロセスコア内部の高速化について示す．図 1 の構成において，A*アルゴリズムにおける $OPEN$ リストをソーティングネットワークを用いて実装しているため，取り出す際に $O(1)$ で f 値が小さい順に複数のノードを取り出すことが可能である．extract 以降の手続きを並列化し，複数のノードから探索して展開されるノードの重複を取り除いた．この詳細を図 4 に示す．extract で取り出すノードの個数が j の場合の構成を $\mathrm{FPGA}_{1,j}$ と表記する．探索されるノードの数が増えるため実行時間の高速化が期待される．しかし，探索するノードが増えるため，ノードを取り出す探索のオーバーヘッドや，通信量が増えるためのオーバーヘッドが生じる．

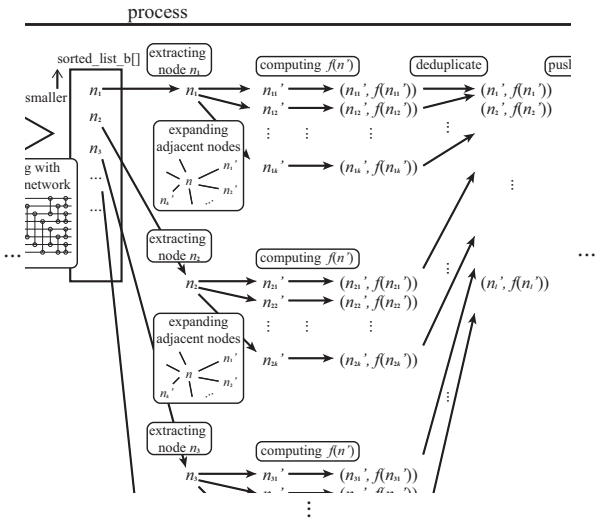

図 4　コア内部の並列化

4.6　実装方法

本研究では Xilinx の FPGA への実装を想定し，Xilinx Vivado 環境を用いて専用ハードウェアの実装を行った．ソーティング，展開，f 値計算，ハッシュ計算を行うプロセス回路については，Vivado HLS を用いて C++ベースで開発を行った．さらに，ループ展開等の並列化について C++コード中にプラグマとして記述した．また，各プロセスを接続するクロスバースイッチに関しては，パケット処理部を Vivado HLS を用いて C++ベースで設計し，周囲のパケット向けバッファや，モジュール間の接続には Block Design を用いて実装を行った．最後に，プロセス回路やクロスバースイッチモジュールを IP 化し，Block Design を用いて全体を記述した．また，Node Table や Graph Table は Block RAM を用いて実装を行った．

5.　専用ハードウェアの評価

評価では Path-finding を解く専用回路を作成し，コアの並列化，展開処理以降を並列化した際の実行時間について評価を示す．初めに並列化を行わない 1 プロセス版の A* 回路を実装し，CPU/GPU との実行時間比較を示す．その後，1 プロセス版の実行結果をベースに回路の並列化を行った場合の実行時間について比較を行う．

5.1　評価条件

専用ハードウェアで解く問題は格子グラフ上のスタートノードからゴールノードまでの最短経路を探索する Path-finding とする．上下左右に隣接するノード間のエッジの重みは 1 とする．このグラフは単調であり，また，斜め 45 度で隣接するノード間にもエッジが存在し，エッジのコストは $\sqrt{2}$ とする．探索するノードに対して上下左右斜めの 8 方向に移動可能とする．ヒューリスティック関数は，斜め移動も含めたマンハッタン距離とする．すなわち，ゴールノード t の座標を (tx, ty)，現在探索するノード n の座標を (x, y) とすると，マンハッタン距離は

$$L1(n,t) = min(tx-x, ty-y) \times \sqrt{2} + max(tx-x, ty-y) - min(tx-x, ty-y) \quad (4)$$

とする．

C++でソフトウェアによる実装をし，ソフトウェアシミュレーション・高位合成・RTL シミュレーションは Xilinx 社の提供するデザインツールである Vivado Design Suite 2019.2.1 で行った．高位合成では，VerilogHDL に変換した．

ハードウェア実装に用いた FPGA は Xilinx Virtex UltraScale+ XCVU9P(VCU118) である．高位合成と回路の合成の際に 100 MHz のタイミング制約を与え，Synthesis の結果を基に RTL シミュレーションを行い，クロック数から実行時間を算出した．

5.2 CPU/GPU との実行時間比較

4 コアでプロセス内並列化を行っていない FPGA 実装 ($FPGA_{4,1}$) と従来手法である 1 コア CPU による実装，GPGPU による実装 [13] とで実行時間を比較した．CPU は Intel® Core™ i7-5820K Processor 3.30GHz，GPGPU は NVIDIA Tesla K20c を用いた．図 5 に問題サイズを 9×9 とした場合，図 6 に問題サイズを 99×99 とした場合の実行時間を示す．FPGA での評価については Vivado HLS が見積もった結果と全体を含んだ RTL シミュレーションから求めた 2 つの結果を示している．$FPGA_{1,1}$(hls) は Vivado HLS が見積もった 1 コア並列化無しの場合の時間を示している．また，$FPGA_{4,1}$(w peripheral) は 4 コア構成で周囲の BRAM やバス等も作りこみ，RTL シミュレーションを行った時間である．

問題サイズが 9×9 の場合は $FPGA_{1,1}$(hls) は 99.925μs となり，CPU や GPU と比較して高速な結果となった．しかしながら，$FPGA_{1,1}$(hls) は Vivado HLS が見積もった時間であり，AXI インタフェースへのアクセスや hls で合成したモジュールのハンドシェイク時間が 0 となっている．メモリアクセスやバスアクセス等を加味した実行時間は $FPGA_{4,1}$(w peripheral) は 1228.5 μs となった．本来は BRAM やバスを含んだ 1 コア版 ($FPGA_{1,1}$(w peripheral)) を評価すべきであるが，計算が正しく行えなかったため結果には示していない．

GPU の計測においてはデータコピー時間を含まないカーネル実行時間の計測を行っているが，4 コア実装であっても FPGA が高速であることがわかる．一方で，GPU の結果は 1core-CPU よりも遅いため，問題サイズに対して並列度が大きいことに起因する探索のオーバーヘッド，GPU 起動オーバヘッドが要因となり，十分な高速化ができていないことが考えられる．

問題サイズが 99×99 の場合は HLS が出力した結果を示している．この結果，GPU や CPU と比較しても高速な結果となった．また，この場合も 9×9 と同様に GPU が 1core-CPU よりも遅い結果となった．同様に並列度の不足や GPU 起動オーバヘッドが原因であると考えられる．本来であれば，さらに問題サイズを大きくした際の評価を行うべきであるが，FPGA 実装の場合，DDR への読み書き回路を追加する必要があり，今後の課題である．

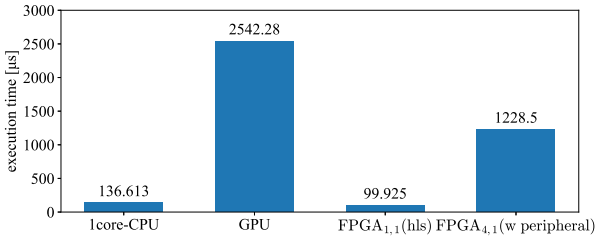

図 5 問題サイズ 9×9

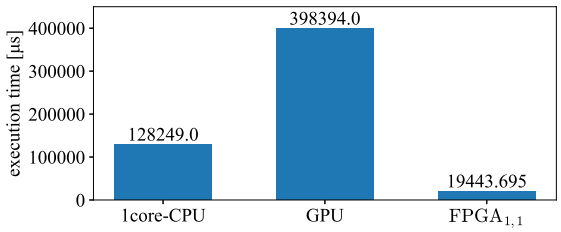

図 6 問題サイズ 99×99

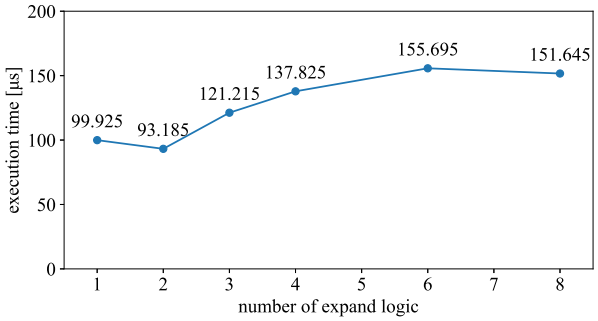

図 7 コア内部の並列度に対する実行時間 (問題サイズ 9×9)

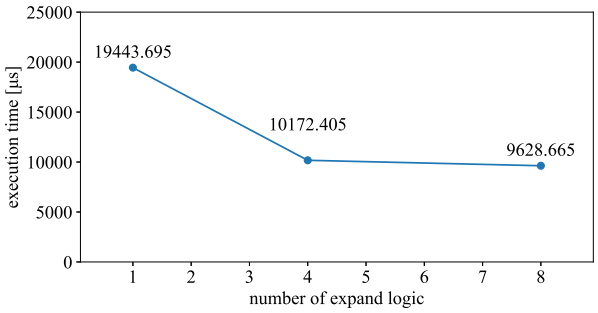

図 8 コア内部の並列度に対する実行時間 (問題サイズ 99×99)

5.3 コア内部の並列化による高速化結果

並列化手法 2 である extract 以降の並列化について，プロセスコアの内部の並列度を変えて実行時間を測定した．こちらの結果も HLS が見積もった時間であるが，コア内部の構成についての並列度については考察することができる．問題サイズが 9×9 の場合は図 7 の結果となった．内部の並列度 2 のときに最速となり，内部の並列化は高速化に貢献することがわかった．並列度が 3 以降は並列度が大きくなるにつれて遅くなったが，これは問題サイズに対して並列度が大きく，探索のオーバーヘッドが生じたからであると考えられる．問題サイズが 99×99 の場合においても，図 8 の結果となり，並列度 8 のときに実行時間が 9628.665 μs となり，高速化に成功した．

5.4 ハードウェア資源量

提案するハードウェアを実際に FPGA に実装し，ハードウェアの資源量の見積もりを行った．1 プロセス版 ($FPGA_{1,1}$(w peripheral)) と 4 プロセス版 ($FPGA_{4,1}$(w peripheral)) において extract 以降の並列化

を行わない場合の結果を示す．

表 2　資源の利用率

	Resource	使用量	XCVU9P に対する利用率
1 コア版	LUT	64,548	5.46%
	FF	44,949	1.90%
	BRAM	47	2.18%
	DSP	13	0.19%
4 コア版	LUT	261,908	22.15%
	FF	183,665	7.77%
	BRAM	244	11.30%
	DSP	52	0.76%

この結果，4 コア版では LUT は 22%近く消費されるが，まだ十分な余裕があることがわかる．また，Vivado の電力レポートでは 1 コア版の消費電力は 2.84W, 4 コア版の消費電力は 3.84W となり，CPU や GPU などと比較すると十分に低いことがわかる．

6. まとめ

本研究では，最短経路探索問題について A*アルゴリズムを用いて解く専用ハードウェアを FPGA を用いて作成した．A*アルゴリズムの実装において，*OPEN* リストをソーティングネットワークで実装し，HDA*アルゴリズムを FPGA 上に実装し，従来手法の 1 コア CPU と GPGPU による実装と比較した．問題サイズが小さい場合はいずれの手法に比べても高速に問題を解くことができた．また，さらなる高速化に向けて，2 つの並列化手法を提案した．1 つは，プロセスコアの並列度を変える方法である．もう 1 つはプロセスコアの内部について，extract 以降を並列化した方法である．いずれの手法においても，実行時間の減少に寄与した．また，ハードウェア資源量について，CPU や GPU などと比較すると十分低いことがわかった．

今後の課題は，FPGA 上に DDR への読み書き回路を追加し，大きいサイズの問題について，実行時間と電力性能についての評価である．さらに，提案した 2 つの並列化手法についても，シミュレーションではなく実機で実行時間や電力性能の測定や大きいサイズの問題においての評価を行いたい．

謝辞　本研究の一部は，JST CREST 課題番号 JPMJCR18K1（研究課題名「エッジでの高効率なデータ解析を実現するグラフ計算基盤」），の支援を受けたものである．

参考文献

[1] Cong, J., Kahng, A. B. and Leung, K. S.: Efficient algorithms for the minimum shortest path steiner arborescence problem with applications to VLSI physical design, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 1, pp. 24–39 (online), DOI: 10.1109/43.673630 (1998).

[2] Endelman, J. B., Silberg, J. J., Wang, Z.-G. and Arnold, F. H.: Site-directed protein recombination as a shortest-path problem, *Protein Engineering Design and Selection*, Vol. 17, No. 7, pp. 589–594 (online), DOI: 10.1093/protein/gzh067 (2004).

[3] Dolgov, D., Thrun, S., Montemerlo, M. and Diebel, J.: Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments, *The International Journal of Robotics Research*, Vol. 29, No. 5, pp. 485–501 (online), DOI: 10.1177/0278364909359210 (2010).

[4] Alexopoulos, C. and Griffin, P. M.: Path Planning for a Mobile Robot, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 22, No. 2, pp. 318–322 (online), DOI: 10.1109/21.148404 (1992).

[5] Dijkstra, E. W.: A note on two problems in connexion with graphs, *Numerische Mathematik*, Vol. 1, No. 1, pp. 269–271 (online), DOI: 10.1007/BF01386390 (1959).

[6] Hart, P. E., Nilsson, N. J. and Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, pp. 100–107 (online), DOI: 10.1109/TSSC.1968.300136 (1968).

[7] Evett, M., Hendler, J., Mahanti, A. and Nau, D.: PRA*: Massively Parallel Heuristic Search, *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, Vol. 25, pp. 133–143 (online), DOI: 10.1.1.46.7864 (1995).

[8] Kishimoto, A., Fukunaga, A. and Botea, A.: Scalable, Parallel Best-First Search for Optimal Sequential Planning (2009).

[9] Bellman, R.: On a routing problem, *Quarterly of Applied Mathematics*, Vol. 16, No. 1, pp. 87–90 (online), DOI: 10.1090/qam/102435 (1958).

[10] Korf, R. E.: Depth-first iterative-deepening. An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, pp. 97–109 (online), DOI: 10.1016/0004-3702(85)90084-0 (1985).

[11] Zobrist, A. L.: A new hashing method with application for game playing, *reprinted in International Computer Chess Association Journal (ICCA)*, Vol. 13, No. 2, pp. 69–73 (1970).

[12] Burns, E., Lemons, S., Ruml, W. and Zhou, R.: Best-first heuristic search for multicore machines, *Journal of Artificial Intelligence Research*, Vol. 39, pp. 689–743 (online), DOI: 10.1613/jair.3094 (2010).

[13] Zhou, Y. and Zeng, J.: Massively Parallel A* Search on a GPU, *In: Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1248–1255 (2015).

[14] Batcher, K. E.: Sorting networks and their applications, pp. 307–314 (1968).

[15] Mueller, R., Teubner, J. and Alonso, G.: Data processing on FPGAs, *Proceedings of the VLDB Endowment*, Vol. 2, No. 1, pp. 910–921 (online), DOI: 10.14778/1687627.1687730 (2009).

[16] Casper, J. and Olukotun, K.: Hardware acceleration of database operations, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*, Association for Computing Machinery, pp. 151–160 (online), DOI: 10.1145/2554688.2554787 (2014).

© 2020 Information Processing Society of Japan