Vol.2011-AL-135 No.2
2011/5/16

超平面ハッシュ関数に基づく分散 A* アルゴリズムの提案と多重配列アラインメント問題への応用

小 林 義 和[†1]　岸 本 章 宏[†1]　渡 辺 　治[†1]

分散メモリ環境での A* アルゴリズムの並列化は，アルゴリズムの解答速度および解答能力向上のために重要な課題である．本論文では，単純であるが効果的な A* の並列化である Hash Distributed A* (HDA*) アルゴリズム上に超平面ハッシュ関数を利用し仕事を分配する方法を提案する．この仕事分配方法の性能を，マルチコア CPU を持つマシンで構成される PC クラスター上で，最適多重配列アラインメントの難しいベンチマーク問題を解かせることで評価し，この手法でベンチマーク問題に対しては，384CPU コアまで高い台数効果を達成すること示した．

A Distributed A* Algorithm Based on the Hyperplane Function and Its Application to Multiple Sequence Alignment

Yoshikazu Kobayashi ,[†1] Akihiro Kishimoto [†1]
and Osamu Watanabe[†1]

Parallelizing the A* algorithm in distributed memory environments plays an important role in both achieving speedups and improving its solving ability. This paper presents a new work distribution strategy based on the hyperplane function, which is combined with simple but scalable parallel A* called Hash Distributed A*. The performance was evaluated on a cluster of multi-core machines and the presented strategy was shown to scale well up to 384 cores in solving difficult instances of the optimal multiple sequence alignment.

†1 東京工業大学 大学院情報理工学研究科 数理・計算科学専攻
Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology

1. 序　　論

最良優先探索は，コンピュータサイエンスにおいて基本的なアルゴリズムであり，その中でも A* アルゴリズム[1) は，カーナビでの経路探索やバイオインフォマティックスにおける遺伝子配列の類似度の計算など，その応用分野は多岐に渡る．

しかし一方で，A* アルゴリズムを，状態数に組み合わせ爆発が生じる広大な探索空間を持つ問題に単純に用いることはできない．膨大な計算時間がボトルネックになる上，さらに，A* は以前に展開したノードをすべて保持するので，膨大な空間を探索しようとするとメモリをすぐに使い切ってしまうからである．

分散メモリ環境で A* を並列化すれば，アルゴリズムの高速化だけでなく，逐次 A* よりもはるかに多くのメモリの利用によって，解答能力も改善できることが期待される．現在，多数の PC がネットワークでつながれている環境が存在するだけでなく，複数の CPU コアを持つ PC も普及しているので，アルゴリズムの並列化は益々重要な研究課題になってきている．さらに，シングル CPU コアあたりの速度改善は今後は頭打ちになると予想されるので，並列化はハードウェアの進歩からの高速化を享受するためには最も効果的な手法である．

Hash Distributed A* (HDA*) は，単純であるが効果的な分散並列 A* アルゴリズムである[2)]．岸本らは，HDA* を人工知能の古典的プランニングに適用し，512 コアの CPU を用いても，速度・解答能力の双方で高い性能が得られることを示した[3)]．しかし，1024 コアを用いたときには，逐次 A* では展開されなかったノードを展開してしまうことが頻繁に起こるため，性能が悪化する場合が観察された.（岸本らの論文では，この余計な探索の増加が起こる原因の分析は詳しく行われなかった.）

A* は汎用的なアルゴリズムであるので，その並列版である HDA* も様々なアプリケーションでの有用性が期待される．その可能性を調べるには，プランニング以外の様々なアプリケーションにも HDA* を適用し，HDA* の利点・欠点を明らかにし，そのボトルネック解析を理論と実験の双方から行うことが必要不可欠である．

本論文では，HDA* を古典的プランニングとは異なる性質を持つ最適多重配列アラインメント問題に適用し，実験的に性能を評価する. まず，この問題の探索を HDA* の枠組みで行うと，展開済みのノードを頻繁に再展開してしまう現象がおき，そのために CPU コア数を増やせば，性能がかえって悪化する場合があることを実験的に示した．そこで各プロセッサに対する仕事の分配方法として，超平面ハッシュ関数に基づく手法（超平面仕事分配法）を提案し，従来の HDA* よりもはるかに高い台数効果を達成できることを実証した．

ⓒ 2011 Information Processing Society of Japan

本論文の構成は次の通りである．2 節では，最適多重配列アラインメント問題を定義し，3 節で A* アルゴリズムについて説明する．4 節で並列探索の性能低下の原因である並列オーバーヘッドについて考察した後，5 節で HDA* アルゴリズムについて概説する．6 節で本論文で提案する超平面仕事分配法について述べ，7 節で実験結果を示す．8 節で関連研究について議論した後，9 節でまとめと今後の課題について述べる．

2. 最適多重配列アラインメント問題

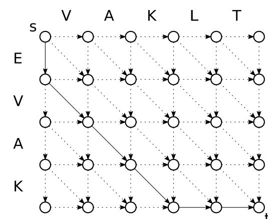

図 1　配列アラインメント問題の例

バイオインフォマティクスで代表的な課題である最適多重配列アラインメント問題は，n 本の DNA の塩基やタンパク質中のアミノ酸の文字列集合を文字列の配列として受け取り，文字列同士を一致させるため，各文字列にギャップを挿入する操作を繰り返し（その際，ペナルティを受ける），最も良いスコアを持つアラインメントを出力する問題である．このスコアは比較する配列の類似度を反映している．

多重配列アラインメント問題は，n 次元のグリッド空間上のある端点から最も遠い端点までの最適な経路を計算する問題として定式化できる．図 1 に，2 本の配列 `VAKLT` と `EVAK` のアラインメントを求める例を示す．このグラフでは，行方向と列方向にそれぞれの配列を並べている．このグリッド空間において，対角線の辺は，各配列の一文字を合わせる操作を表し，水平および垂直の辺はどちらかの配列にギャップを挿入することに対応する．各辺にはコストが割り当てられており，そのコストは対応する 2 つの文字の類似度やギャップ挿入のペナルティを考慮して前もって定められている．最適なアラインメントを求めることは，左上の頂点 s から右下の頂点 t への最小の合計コストを持つ経路を求めることに等しい．図 1 の実線はこのグラフでの最適な経路であり，s から t への太字の経路は `-VAKLT` と `EVAK--` という最適アラインメントを表す．

3. A* アルゴリズム

多重配列アラインメント問題のように，初期ノードから目標ノードまでの最適なコストを持つ経路を求める問題は，A* アルゴリズム[1)]でグラフ探索を行えば最適解を求められる．

A* アルゴリズムでは，探索の枝刈りにヒューリスティック関数 $h(\cdot)$ を用いる．$h(u)$ は，ノード u に到達した時点で，u から目標ノード t に至るまでにかかる合計コストを推定する関数である．通常は，解の最適性を保証するために許容的な (admissible) ヒューリスティック関数が使われる．任意の u に対して，$h(u)$ が u から t までの実際の合計コストの最小値よりも大きく見積もることがないとき，関数 $h(\cdot)$ を許容的という．A*は，同じヒューリスティック関数を利用するアルゴリズムの中で，ノードを展開する回数が最も少ないアルゴリズムであることが知られている．

初期ノード s からノード u に至るまでにかかった合計コストを g-値と呼び，$g(u)$ と表わす．s から u を経由して目標ノードに至る合計コストの推測値を f-値と呼び，$f(u)$ と表わす．つまり，$f(u) := g(u) + h(u)$ と定義する．A* は，オープンリストとクローズドリストを利用して，最適解を求めるまでノードの展開を繰り返す最良優先探索である．オープンリストは，未展開ノードのすべてを f-値の小さな順に並べるデータ構造である．一方，クローズドリストは，すでに展開したノードすべてを保持するデータ構造であり，探索空間がグラフ構造であるときに，新たに生成されたノードがすでに展開済みであるかの判定と，A* が解を見つけたときの最適経路の計算に利用する．$cl(v)$ をクローズドリストに保存されたノード v の f-値とすると，A* の代表的な実装手法では次の動作を繰り返す：

(1) オープンリストに初期ノード s を格納する．
(2) オープンリストから f-値が最小のノード u を取り出す．オープンリスト空のときには，解は存在しない．
(3) u が目標ノード t であれば，$g(u)$ と s から u に至る経路を返す．
(4) そうでなければ，u を展開し，u の子ノード v すべてを生成し，各 v について，次を行う．
　(a) v がクロードリストにあり，$cl(v) \leq f(v)$ ならば，v はオープンリストに格納しない．
　(b) v がクローズリストにあり，$cl(v) > f(v)$ ならば，$cl(v) = f(v)$ に更新し，v をオープンリストに入れ直す．
　(c) それ以外のときには，v をオープンリストに入れる．

ⓒ 2011 Information Processing Society of Japan

(5) u をクローズドリストに格納する．
(6) (2)-(5) を繰り返す．

許容性から，$f(u)$ の値が，初期ノードからノード u を経由して目標ノードに至るまでにかかる合計コストを上回ることはない．さらに，A* は f-値の小さな順番に節点を展開するので，A* が目標ノードをまだ見つけていない場合，最適解の合計コストは，必ずオープンリストの先頭にあるノードの f-値以上であることが証明できる．この性質より，A* が目標状態を見つけたときには，必ず最適解であることが保証される．

本論文では，A* が利用する $h(\cdot)$ を単調 (consistent) であるとする．$h(\cdot)$ が単調であるとは，任意のノード u に対して，次が成立することである：

(1) u が目標ノードならば，$h(u) = 0$,
(2) それ以外の時は，$h(u) \leq c(u,v) + h(v)$．ただし，v は u の子ノードで，$c(u,v)$ は u, v 間の辺のコストである．

単調な $h(\cdot)$ は許容的であるだけでなく，A*アルゴリズムの (4) でノード v がすでにクローズドリストにあるときには常に $cl(m) \leq f(m)$ が成立することが保証できる．(以下では，v がクローズドリストにあり $cl(v) \leq f(v)$ が成立することを「ノード v の重複判定に成功する」と呼ぶ.）つまり，v がいったんクローズドリストの保存されれば，v を再びオープンリストに入れること（以下「再オープン」と呼ぶ）は起きない．

4. 並列オーバーヘッド

並列探索により高い台数効果を達成する並列探索アルゴリズムの開発は一般には難しい．逐次探索には存在しない並列オーバーヘッドの問題が生じる場合が多いからである．本論文が対象とする A* を並列化すると，主に次の 3 つのオーバーヘッドが生じる可能性がある：

- 探索オーバーヘッド：並列探索のみで生じる余計なノード展開であり，次に定義される値で量的に評価される．

$$\text{SO} := \frac{\text{並列アルゴリズムによる合計ノード展開数}}{\text{逐次アルゴリズムでのノード展開数}} - 1$$

- 同期オーバーヘッド：並列探索の順番に依存関係があるとき，あるプロセッサが他のプロセッサが行う探索の終了を待つ際に生じるアイドル時間である．以下に定義されるロードバランスは，各プロセッサに仕事が均等に割り当てられているかを表わす量だが，これは同期オーバーヘッドが増加する原因を分析するためにも利用できる．

$$\text{LB} := \frac{\text{各プロセッサによるノード展開数の最大値}}{\text{各プロセッサによるノード展開数の平均値}}$$

- 通信オーバーヘッドは：プロセッサ間で仕事を分割したり，探索結果を共有するときなどに生じる通信遅延である．

これらの 3 つのオーバーヘッドは相互に密接した関係にあり，これらのオーバーヘッドの組み合わせを最小化すれば高い台数効果が得られる．しかし，理論的に最適な組み合わせを見つけることは難しいので，現実的な並列探索アルゴリズムの設計では，実験的にオーバーヘッドの少ない手法を開発することが多い．

5. Hash Distributed A* (HDA*) アルゴリズム

単純であるが効果的な分散並列 A* アルゴリズムとして提案されたのが HDA* アルゴリズム[2] である．HDA* では，各プロセッサはプロセッサ間で互いに素な形でクローズドリスの一部を管理し，全体で一つの大きなクローズドリストとみなせるデータ構造を保持する[⋆1]．オープンリストに関しても同様である．このため，分散メモリ環境の利用により増加したメモリを，プロセッサ間で重なりのないクローズおよびオープンリストに割り当てられるので，HDA* はメモリのスケーラビリティが良い．

文献[5] のように，HDA* では，各プロセッサ P_i は自分自身のオープンリストで f-値が最小のノードを選択し展開する．P_i が新たに生成した子ノード v は，v をクローズドリスト（またはオープンリスト）に保持すべきプロセッサ P_j に送信される．HDA* では，v が以前に展開済みであれば，P_j のクローズドリストに必ず $cl(v)$ が保存されているので，$f(v)$ との比較を行い，v を再オープンするべきかどうかが決定できる．また，文献[6] のように，P_j にノードを送った後には，P_i は直ちに別ノードを展開できるので，プロセッサ間の同期が不要である．なお，各プロセッサは，定期的に新しいノードが届いているかどうかを確認する．HDA* は，こうした計算を，発見した解の最適性が確認されるまで繰り返す．

プロセッサ数が多くなれば，HDA* は高い確率で別プロセッサに生成したノードを送信すると考えられるので，通信オーバーヘッドを減らすことは HDA* の性能を上げるのに重要である．このため，HDA* ではノードを生成するたびに送信するのではなく，プロセッサ P に送るメッセージが k 個以上になった場合に，まとめて送信するのが通常である．この k の選択は，対象問題や計算機環境に依存する．

各ノードを保持すべきプロセッサの決定にはハッシュ関数を利用する．HDA* は，ほぼ

⋆1 HDA*とほぼ同一のアルゴリズムは文献[4] にも存在する．ただし，HDA* とは異なるハッシュ関数を利用し，さらに複雑な処理を付加している．

© 2011 Information Processing Society of Japan

一様にハッシュ値が分布する Zobrist 関数[7] を利用しているので，本論文でもこの関数を利用する．多重配列アラインメント問題で，l_i を i 番目の配列の長さとすると，ノード $\boldsymbol{x}$ は $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ と表現できる（ただし，$0 \leq x_i \leq li$）．また R_i を $l_i + 1$ 個の要素を持つ，事前に初期化された乱数表とする．ここでは Zobrist 関数 Zobrist を，以下のように定義する：

$$\mathrm{Zobrist}(\boldsymbol{x}) := R_1[x_1] \text{ xor } R_2[x_2] \text{ xor } \cdots \text{ xor } R_n[x_n]$$

6. 超平面ハッシュ関数に基づく HDA*アルゴリズム

6.1 HDA*の欠点

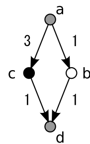

図 2 HDA*が非最適な順序でノードを展開した場合に，ノードを再オープンしてしまう例

単調な $h(\cdot)$ を用いた A* では，ノード u がいったんクローズドリストに保存されれば，別経路からの u が新たに作られた場合には u を再オープンする必要がない．しかし，HDA* では，大局的に最小の f-値を持つノードを展開するわけではないので，単調な $h(u)$ でも $cl(u) > f(u)$ が成立する場合があり，u を再オープンし，再展開してしまうことがある．

岸本らが利用した古典的プランニング問題では辺のコストが均一だった．それに対し，多重配列アラインメントでは辺のコストが均一ではなく，同一ノードに至る経路が非常に多いので，クローズドリストに保存されたノードの再オープンはより頻繁に起こる．

図 2 を利用して，ノードの再オープンが起こる状況を説明する．HDA* において，ノード a と d はプロセッサ P_1 が展開し，b は P_2，c は P_3 が展開すると仮定する．すると，以下のような順でノードを展開されたとき，P_1 が d を再オープンすることになる：

(1) P_1 は a を展開し，b と c を生成する．b は P_2 へ，c は P_3 へと送信される．
(2) P_3 は c を展開し，d を生成して P_1 へと送る．
(3) P_1 は d を展開し，d を P_1 のクローズドリストに加える．このとき，$g(d) = 3+1 = 4$ となる．
(4) P_2 は b を展開し，d を生成して P_1 へと送信する．
(5) P_1 は d を受信するが，このときの $g(d)$ は $1 + 1 = 2$ である．一方，$cl(d)$ の保存時の $g(d)$ は 4 なので，$cl(d) > f(d)$ となるため，P_1 は d を再オープンする．

各辺のコストが均一な場合には，このようなノードの再オープンは発生しにくい．図 2 では，経路 $a \to b \to d$ と $a \to c \to d$ の長さは等しいので，$a \to c \to d$ で d に至る場合のみを探索すれば，解の最適性が保証される．

6.2 超平面仕事分配

本節では，Zobrist 関数に基づく HDA* で生じるノードの再オープンの問題をなるべく回避するために，超平面仕事分配法を提案する．この手法では，ノード $\boldsymbol{x}$ を展開すべきプロセッサを求める新しい関数（超平面ハッシュ関数）を以下のようにを定める：

$$\mathrm{Plane}(\boldsymbol{x}, d) := \begin{cases} \lceil \frac{1}{d} \sum x_i \rceil & \left(d \in \{1, 2, 3, ...\}\right) \\ \frac{1}{d} \sum x_i + \left(\mathrm{Zobrist}(\boldsymbol{x}) \bmod \frac{1}{d}\right) & \left(d \in \{\frac{1}{2}, \frac{1}{3}, ..., \frac{1}{p}\}\right) \end{cases}$$

ただし，p は実行プロセッサ数である．d は超平面の厚さであり，後に述べるように経験的に決める．この上で，$P(\boldsymbol{x}) := Plane(\boldsymbol{x}, d) \bmod p$ と定義し，ノード $\boldsymbol{x}$ に対する値 $P(\boldsymbol{x}) = i$ のとき，このノードをプロセッサー P_i に割り当てる（定義より，$0 \leq i < p$ となる）．

超平面仕事分配法では，Zobrist 関数に基づく仕事分配と同様に，いったん $\boldsymbol{x}$ が P_i に送られれば，P_i が $\boldsymbol{x}$ の重複判定を行う．さらに，n 次元の多重配列アラインメントでは，$\boldsymbol{x}$ が生成する子ノード集合を $\mathrm{Children}(\boldsymbol{x})$ とすると，超平面仕事分配では次の性質が成り立つ：

定理 1

$$\#\left(\bigcup_{x\,:\,P(x)=i} \{P(\boldsymbol{x}') \mid \boldsymbol{x}' \in \mathrm{Children}(\boldsymbol{x})\}\right) \leq \left\lfloor \frac{n}{d} + \max(1, \frac{1}{d}) \right\rfloor$$

定理 1 は，超平面仕事分配では各プロセッサが各ステップで生成する子ノードが所属するプロセッサ数が高々 $\lfloor n/d + \max(1, 1/d) \rfloor$ であることを保証している．つまり，この数は d を調整すれば，p と n が大きな場合には，$\min(p, 2^n - 1)$ に比べて非常に小さくできるので，複数の子ノードが同じプロセッサに割り当てられる可能性が高くなる．同じプロセッサに複数の子ノードが所属すれば，これらの子ノード間では有望な順番にノードを展開できるので，不必要なノードの再展開を防ぐことにつながる．

図 2 で，b と c の両方が P_2 に割り当てられ，かつ $h(b) = h(c)$ であると仮定する．$g(b) < g(c)$ なので，P_2 は c よりも b を先に展開できる．この場合，d は経路 $a \to b \to d$ で先に到着するので，d を再オープンしない．

さらに，超平面仕事分配は，Zobrist 関数による仕事分配よりもノードを展開してから送信するまでの時間が短い．5 節で述べたように，HDA*ではプロセッサ P が Q にノードを送

© 2011 Information Processing Society of Japan

信するときには，k 個のノードを一つのメッセージにまとめてから送信する．もし，Zobrist 関数が P が生成するノードを各プロセッサに等確率で分配すると仮定すれば，あるノードが Q に所属する確率は $\frac{1}{p}$ である．しかし，超平面仕事分配では，定理 1 より，この確率は $\frac{1}{p}$ よりもはるかに大きいからである．

超平面仕事分配法では，ノードを割り当てるプロセッサを制限するので，Zobrist 関数よりもロードバランスが悪くなるという欠点があるが，d を調整すれば対応できる．実際，d を $\frac{1}{p}$ を近づければ，超平面仕事分配法は，Zobrist 関数による分配に振る舞いが近づく．

適切な d の選択は，超平面仕事分配法が高い性能を発揮するためには重要である．しかし，最適な d は対象となる問題に依存する場合が多いので，どのような場合にも効果的なロードバランスを達成するような d の計算法は難しいだろう．本論文では，d を適切に選択する基準として，配列の長さの合計値 l とプロセッサ数 p を用いたものが利用できると考え，いくつかの関数を実験的に調べた結果，d を以下のように定義した：

$$d := \begin{cases} \mathrm{round}(\frac{\lambda l}{\log p}) & (\frac{\lambda l}{\log p} \geq 1) \\ (\mathrm{round}(\frac{\log p}{\lambda l}))^{-1} & (\frac{\lambda l}{\log p} < 1) \end{cases}$$

ここで $\mathrm{round}(x)$ は四捨五入であり，λ は実験で定める定数である．実際には，いくつかの多重配列アラインメント問題を解き事前に台数効果を調べることで，$\lambda = 0.003$ に設定した．

7. 実験結果

7.1 実験条件

前節までに述べた逐次および並列 A* を実装し，東京工業大学の TSUBAME2.0 上で性能比較を行った．TSUBAME2.0 の各計算ノードには，54GB のメモリと 12 コアの CPU（2.93GHz, 6 コア × 2）があり，Quad Data Rate Infiniband × 2 (80Gbps) により接続されている．実験では，32 計算ノード（合計 384 コア）まで利用した．並列 A* は，Intel C コンパイラでコンパイルし，代表的な MPI ライブラリーである MVAPICH2[⋆1] を用いて並列化した．さらに並列 A* では，いくつかの初期実験を行った結果，送信の際に一括してデータ送るノード数は 256 とした．

逐次および並列 A* のヒューリスティック関数 $h(\cdot)$ には，多重配列アラインメント問題でよく用いられる，最適ペアワイズアラインメントのスコアの合計値を用いた．つまり，あらかじめすべてのペアワイズアラインメントの組み合わせを計算し，その結果をテーブルに保存し，実行時にはテーブル参照で $h(\cdot)$ を求めた．したがって，$h(\cdot)$ の計算時間は格段に短い．また，コスト関数には，この分野で代表的な PAM250[⋆2] に，ギャップコストとして最小値の -8 を与えたものを用いた．ただし，A* では各辺のコスト値が非負の場合にしか最適解を求められないため，すべて正の値になるように値を嵩上げした．

多重配列アラインメント問題では，最適解の上界値 ub の計算は，最適解の計算よりもはるかに簡単である．さらに，A* にこの ub を与え，$f(v) > ub$ を満たす v をオープンリストに保存しなければ，A* の解答能力向上につながる．本論文では，文献[8] に基づき，ub の計算に $ub(v) = g(v) + w \cdot h(v)$（ただし $w > 1$）を利用し，目標ノードを見つけるまで $ub(v)$ の小さな順番に探索を行い，最適解の w 倍以下の上界値を返せる Weighted A* アルゴリズムを用いた．本論文では，$w = 1.02$ とした．逐次および並列 A* は，この ub を受け取った後に，探索を行った．

ベンチマーク問題集には，現実のタンパク質アラインメントデータベースの中で代表的な BAliBASE3.0[9] にあるタンパク質配列集合を用いた．BAliBase3.0 にある難しい問題での並列化の効果を確かめるため，逐次 A* で 10 分間以上 1 時間未満で解けた 7 題，そして Zobrist 関数を用いた HDA* が 12 コアで，1 時間未満で解けるが逐次 A* では解けない 4 題を例題として用いた．

7.2 結果と考察

表 1 逐次 A*による各問題のの実行時間

問題名	配列数	展開ノード数	実行時間	初期化時間の割合
BB12022	5	166,237,332	1900.66	0.27 (1.410e-04)
BBS12023	5	251,495,502	3026.12	19.01 (6.243e-03)
BBS11037	5	324,063,825	3399.74	47.72 (1.384e-02)
BBS11026	7	13,097,629	312.05	6.28 (1.973e-02)
BB12036	7	64,729,569	1849.95	1.85 (1.001e-03)
BBS12010	7	65,595,366	2967.83	0.37 (1.232e-04)
BBS12032	9	4,020,764	446.41	2.55 (5.681e-03)

表 1 に逐次探索の性能を示す．初期化では，$h(\cdot)$ に必要な前計算であるペワイズアラインメントのスコアテーブルの生成と上界値の計算を行った．その時間の全体の計算に対する割合が初期時間の割合である．今回に利用した問題では，初期化に要した時間は全体に比べて小さいので逐次計算でを行い，並列化の性能計測の際には，初期化時間は無視した．初期化に時間がかかるときには，この部分の並列化も必要であるが，今後の研究課題としたい．

⋆1 http://mvapich.cse.ohio-state.edu

⋆2 http://prowl.rockefeller.edu/aainfo/pam250.htm

© 2011 Information Processing Society of Japan

表 2 Zobrist 関数を用いた HDA* の台数効果（p: CPU コア数，n: 配列数）

name n	$p=12$ time (speedup)	SO	idle time	LB	$p=24$ time (speedup)	SO	idle time	LB	$p=48$ time (speedup)	SO	idle time	LB
BB12022 5	218.19 (8.71)	0.25	4.18	1.21	106.07 (17.92)	0.25	1.24	1.14	54.31 (34.99)	0.21	1.30	1.18
BBS12023 5	300.44 (10.07)	0.24	7.71	1.17	146.74 (20.62)	0.14	6.78	1.22	75.06 (40.32)	0.16	3.88	1.18
BBS11037 5	311.67 (10.91)	0.09	1.97	1.20	164.31 (20.69)	0.15	2.67	1.13	85.11 (39.94)	0.14	3.12	1.15
BBS11026 7	29.57 (10.55)	0.04	0.75	1.04	14.69 (21.25)	0.05	0.00	1.04	7.74 (40.31)	0.05	0.20	1.04
BB12036 7	175.08 (10.57)	0.04	7.22	1.04	86.26 (21.45)	0.04	1.90	1.05	44.80 (41.29)	0.04	1.81	1.04
BBS12010 7	288.49 (10.29)	0.16	0.00	1.14	140.09 (21.19)	0.14	2.01	1.11	69.30 (42.82)	0.13	0.48	1.11
BBS12032 9	39.33 (11.35)	0.01	0.47	1.00	20.19 (22.11)	0.01	0.51	1.00	10.18 (43.83)	0.01	0.22	1.01
BB12023 5	694.53 (–)	–	16.75	1.18	297.81 (–)	–	13.07	1.18	140.07 (–)	–	4.02	1.17
BB12003 8	970.38 (–)	–	25.26	1.04	478.98 (–)	–	13.36	1.03	237.87 (–)	–	2.72	1.03
BBS12005 9	631.41 (–)	–	19.31	1.03	311.99 (–)	–	7.97	1.04	157.94 (–)	–	0.77	1.04
BB12032 9	1709.26 (–)	–	54.00	1.01	844.25 (–)	–	13.48	1.01	428.21 (–)	–	9.06	1.01

name n	$p=96$ time (speedup)	SO	idle time	LB	$p=192$ time (speedup)	SO	idle time	LB	$p=384$ time (speedup)	SO	idle time	LB
BB12022 5	30.25 (62.84)	0.23	2.53	1.14	14.99 (126.82)	0.32	0.78	1.12	>600.00 –	–	–	–
BBS12023 5	39.21 (77.18)	0.19	1.50	1.15	20.40 (148.33)	0.29	0.90	1.12	>600.00 –	–	–	–
BBS11037 5	42.71 (79.59)	0.12	1.04	1.11	21.84 (155.67)	0.14	1.00	1.10	>600.00 –	–	–	–
BBS11026 7	4.98 (62.72)	0.14	0.26	1.05	3.43 (91.05)	0.26	0.40	1.07	34.98 (8.92)	4.11	14.63	1.16
BB12036 7	23.46 (78.87)	0.02	0.97	1.04	48.27 (38.33)	0.60	12.04	1.61	64.87 (28.52)	1.23	20.90	1.38
BBS12010 7	41.02 (72.36)	0.27	0.88	1.11	87.26 (34.01)	1.87	16.41	1.26	79.76 (37.21)	2.11	32.06	1.29
BBS12032 9	5.70 (78.35)	0.02	0.22	1.02	15.76 (28.32)	0.41	4.25	1.11	9.53 (46.87)	1.55	3.93	1.16
BB12023 5	73.73 (–)	–	4.92	1.14	39.78 (–)	–	4.96	1.13	>600.00 (–)	–	–	–
BB12003 8	123.00 (–)	–	3.69	1.04	139.30 (–)	–	23.82	1.39	135.69 (–)	–	31.01	1.27
BBS12005 9	82.36 (–)	–	0.43	1.04	51.70 (–)	–	6.24	1.16	99.00 (–)	–	20.46	1.81
BB12032 9	264.09 (–)	–	21.75	1.10	157.44 (–)	–	26.60	1.16	162.67 (–)	–	33.53	1.26

表 2 に従来の手法である Zobrist 関数による仕事分配を行う HDA* 実行性能を示す．idle time は全プロセッサの平均アイドル時間を表す．Zobrist 関数では，96 CPU コアまでは良い台数効果が得られ，96 コアの高速化率は 63 倍から 80 倍 であった．しかし，192 コアでは，96 コアに比べて実行時間が低下する問題が 4 題存在した．さらに，384 コアの場合ではほとんどの問題に対して，192 並列と比較して速度低下が見受けられただけでなく，5 つの配列のアラインメントを求める 4 問では，600 秒間プログラムを実行しても解けなかった．100 コア以上で性能が低下した問題の多くでは，この手法では探索オーバーヘッドが上昇し，アイドル時間も増えた．さらに，CPU コア数を増やすほど，ロードバランスも悪くなる傾向があっただけでなく，この表では省略しているが，通信オーバーヘッドの増加も確認した．

表 3 は，超平面仕事分配による HDA* の性能である．超平面仕事分配では，Zobrist 関数よりもはるかに高い台数効果が得られた．この手法では，192 コア以上では，どの問題に対しても Zobrist 関数よりも実行時間が短く，384 コアでは 123 倍から 322 倍の高速化率を達成した[⋆1]．両手法で解ける問題に対しては，超平面仕事分配法は Zobrist 関数よりもロードバランスが悪い傾向が見られた．ロードバランスの悪ければ，平均アイドル時間も増加した．

超平面仕事分配法は Zobrist 関数よりも効果的であったが，今回提案した超平面の厚さ d の計算方法は必ずしも最適なものではない．例えば，BB12036 を 384 コアで解いた場合には，本手法では $d=1$ に設定し，123 倍の高速化率を達成している．しかし，実際には $d=\frac{1}{4}$ にすれば 254 倍の高速化率になった．より適切な d の計算方法の開発は今後の課題である．

表 4 は，両手法が一度クローズドリストに入れられたノードを再オープンした回数である．Zobrist と Plane は，それぞれ Zobrist 関数と超平面仕事分配法を表す．96 コアでは，両手法の再オープンの回数は同じくらいである．しかし，192 コアでは 2 題を除いて超平面仕事分配の方が再オープン回数が少なく 384 コアではすべての問題で超平面仕事分配の方が再オープンを引き起す頻度が少なかった．よって，超平面仕事分配は，クローズドリストにあるノードの再展開の頻度を減らすことで，探索オーバーヘッドを減らせたと言える．

⋆1 一部の問題では，理論値よりも大きな高速化率を達成している．この原因の一つとして，逐次 A* の方が並列 A* よりもキャッシュ・ミスが多いことが推測される．我々のオープンおよびクローズドリストの実装には，固定長のハッシュ表を用いているが，各計算ノードあたりに割り当てるハッシュ表の大きさを同一にしている．そのため，1 コアあたりに割り当てられるハッシュ表の大きさは，逐次 A* の方が並列 A* よりも大きい．さらに，TSUBAME2.0 の各計算ノードは，ヘキサコア CPU ごとに L2 キャッシュを保持しているので，2 つのヘキサコア CPU を利用する並列 A* の方が逐次 A* よりも各計算ノードあたりのキャッシュサイズが大きい．キャッシュとハッシュ表の大きさの関係を示唆する事実として，ハッシュ表へのノード保存の際に衝突が起こりにくい簡単な問題に対して，逐次 A* のハッシュ表のサイズを変更し，プロファイラを用いてハッシュ表への参照速度を調べた結果，ハッシュ表を大きくすれば参照速度が低下することを確認した．

ⓒ 2011 Information Processing Society of Japan

表 3　超平面仕事分配による HDA* の台数効果（p: CPU コア数，n: 配列数，l: 配列の長さの合計）

name n, l	$p = 12$ time (speedup)	SO	idle time	LB	$p = 24$ time (speedup)	SO	idle time	LB	$p = 48$ time (speedup)	SO	idle time	LB
BB12022 5, 1280	207.04 (9.18)	0.26	0.75	1.28	108.71 (17.48)	0.29	3.17	1.31	51.32 (37.04)	0.29	0.41	1.28
BBS12023 5, 3481	280.52 (10.79)	0.19	6.40	1.19	144.55 (20.94)	0.27	5.15	1.19	70.35 (43.01)	0.27	4.15	1.26
BBS11037 5, 1870	325.26 (10.45)	0.19	5.89	1.16	157.65 (21.56)	0.19	3.97	1.14	80.09 (42.45)	0.17	2.46	1.19
BBS11026 7, 604	30.12 (10.36)	0.02	0.59	1.04	15.58 (20.03)	0.07	0.00	1.04	7.74 (40.29)	0.04	0.16	1.05
BB12036 7, 1499	169.91 (10.89)	0.02	3.21	1.05	87.97 (21.03)	0.03	2.27	1.06	44.09 (41.96)	0.03	2.84	1.04
BBS12010 7, 2221	261.03 (11.37)	0.13	0.00	1.08	130.86 (22.68)	0.14	0.03	1.12	65.23 (45.49)	0.13	2.10	1.15
BBS12032 9, 586	40.99 (10.89)	0.01	0.99	1.01	21.03 (21.23)	0.01	0.49	1.00	10.59 (42.16)	0.01	0.25	1.01
BB12023 5, 3593	567.77 (–)	–	6.91	1.17	286.19 (–)	–	10.52	1.24	133.43 (–)	–	8.56	1.26
BB12003 8, 586	956.58 (–)	–	10.81	1.04	492.05 (–)	–	5.19	1.03	244.59 (–)	–	5.47	1.04
BBS12005 9, 1815	610.53 (–)	–	6.74	1.03	310.29 (–)	–	8.05	1.05	153.60 (–)	–	3.83	1.04
BB12032 9, 613	1688.52 (–)	–	15.71	1.01	880.70 (–)	–	12.48	1.01	442.14 (–)	–	11.16	1.01

name n, l	$p = 96$ time (speedup)	SO	idle time	LB	$p = 192$ time (speedup)	SO	idle time	LB	$p = 384$ time (speedup)	SO	idle time	LB
BB12022 5, 1280	23.77 (79.95)	0.25	0.89	1.36	12.17 (156.19)	0.28	0.74	1.30	6.75 (281.75)	0.36	0.36	1.23
BBS12023 5, 3481	32.32 (93.63)	0.18	1.45	1.23	16.52 (183.22)	0.20	1.86	1.24	10.50 (288.34)	0.27	3.03	1.43
BBS11037 5, 1870	38.04 (89.37)	0.20	0.99	1.20	19.12 (177.79)	0.20	1.16	1.21	10.54 (322.67)	0.18	1.71	1.31
BBS11026 7, 604	4.28 (72.88)	0.05	0.15	1.05	2.45 (127.15)	0.12	0.23	1.07	1.86 (168.12)	0.23	0.51	1.18
BB12036 7, 1499	25.92 (71.36)	0.06	5.15	1.16	15.88 (116.49)	0.07	5.44	1.37	15.03 (123.11)	0.11	9.57	2.39
BBS12010 7, 2221	32.36 (91.72)	0.11	1.81	1.11	16.67 (178.06)	0.24	0.47	1.14	10.14 (292.56)	0.47	1.18	1.18
BBS12032 9, 586	6.70 (66.58)	0.03	1.08	1.06	3.64 (122.58)	0.04	0.74	1.06	2.68 (166.75)	0.07	1.02	1.33
BB12023 5, 3593	62.91 (–)	–	3.54	1.24	32.96 (–)	–	4.52	1.23	21.78 (–)	–	7.57	1.59
BB12003 8, 586	175.67 (–)	–	38.50	1.11	66.00 (–)	–	3.01	1.07	42.31 (–)	–	10.31	1.31
BBS12005 9, 1815	79.72 (–)	–	5.73	1.03	42.22 (–)	–	1.57	1.05	33.78 (–)	–	9.80	1.30
BB12032 9, 613	221.28 (–)	–	8.91	1.02	117.66 (–)	–	9.50	1.05	74.97 (–)	–	19.85	1.26

表 4　クローズドリストにあるノードが再オープンされた回数

name	n	distribution	$p = 96$ reopen	$p = 192$ reopen	$p = 384$ reopen
BB12022	5	ZOBRIST PLANE	**26,931,867** 31,406,029	45,206,738 **34,649,396**	– **38,539,535**
BBS12023	5	ZOBRIST PLANE	37,192,810 **29,155,948**	55,657,404 **32,080,412**	– **36,633,254**
BBS11037	5	ZOBRIST PLANE	**37,750,926** 61,706,272	**45,495,779** 63,877,579	– **57,623,110**
BBS11026	7	ZOBRIST PLANE	1,389,909 **139,358**	3,010,496 **296,010**	52,432,014 **823,296**
BB12036	7	ZOBRIST PLANE	**791,046** 1,016,997	34,342,573 **1,544,408**	75,757,905 **3,479,174**
BBS12010	7	ZOBRIST PLANE	12,153,632 **1,265,006**	85,180,615 **1,507,513**	107,473,282 **2,237,737**
BBS12032	9	ZOBRIST PLANE	**57,683** 100,910	1,603,316 **115,733**	6,203,252 **243,561**
BB12023	5	ZOBRIST PLANE	**43,100,914** 75,639,061	**68,259,973** 73,777,509	– **66,966,386**
BB12003	8	ZOBRIST PLANE	**292,734** 8,779,841	20,701,650 **443,640**	40,359,186 **2,027,297**
BBS12005	9	ZOBRIST PLANE	240,489 **164,489**	884,404 **718,479**	24,526,060 **1,067,810**
BB12032	9	ZOBRIST PLANE	4,446,211 **288,224**	5,857,432 **725,350**	14,637,606 **3,063,362**

8. その他の関連研究

本節では，多重配列アラインメントのような複数経路で同一ノードに至る頻度の高い探索空間での，逐次および並列探索に関する関連研究について述べる．

IDA* アルゴリズム[10] では，最良優先探索である A*を反復深化に基づく深さ優先探索に変換し，探索の深さに比例する大きさのメモリされあれば動作できるようにしている．しかし，IDA* には，以前に探索したことのあるノードを再展開するという欠点があり，これが解答速度の低下につながる．特に，IDA* は複数経路で同一ノードに至るたびにそのノードを再展開してしまう．一方，単調な $h(\cdot)$ を用いる A* ではこのような再展開は起こらない．

Frontier A* (FA*) アルゴリズム[11] は，オープンリストのみを保持することにより，A* のメモリ使用量を削減している．FA* では，クローズドリストなしで複数経路による同一ノードの再オープンを防ぐために，$h(\cdot)$ を単調なものに限り，各ノードに禁止フラグと呼ばれる構造体を追加している．FA* の禁止フラグによる再オープンの削減には，逐次 A* と単調な $h(\cdot)$ で理論的に保証できる探索順序を利用しており，HDA* ではこの性質が成立しない．

PFA*-DDD[12] は，FA* と Delayed Duplicate Detection (DDD)[13] を組み合わせたアルゴリズムを並列化している．しかし，PFA*-DDD では，仕事分配処理の負荷が高く，多くの同期ステップが存在することが指摘されている[14]．一方，超平面仕事分配法は HDA*の性質を

ⓒ 2011 Information Processing Society of Japan

保持するので，ノードを展開する際には同期をとる必要がない．

Burns らは，親子関係または親と子孫の関係にあるノードのいくつかを要約化し，要約化したノードは，同一のコアが探索することで，共有メモリ環境のスレッド間で生じる同期オーバーヘッドを減らしている[14]．各プロセッサが局所的な仕事を行うという意味では，要約化は超平面仕事分配と似ているが，Burns らの研究は共有メモリ環境に限られている．

Mahapatra と Dutt の LOHA では，巡回セールスマン問題を解く際に，探索空間を互いに素な形で分割し，制限された複数のプロセッサがその分割した空間を探索している[4]．超平面仕事分配と LOHA は，ハッシュ関数を利用して仕事を送信するプロセッサを制限する点が似ている．しかし，その目的とするところが大きく違う．LOHA の目的は，その当時に流行していたハイパーキューブ型計算機に存在する深刻な通信オーバーヘッドを減らす目的でハッシュ関数を利用している．この環境では，サブキューブと呼ばれるプロセッサ群が存在するが，サブキューブ間の通信はサブキューブ内の通信よりも遅い．そのため，LOHA の仕事分配戦略では，サブキューブに粒度の粗い仕事をまず分割し，この仕事をサブキューブ内でさらに分割している．これに対し，超平面ハッシュ関数は，現在の分散並列計算機環境での問題点解消が目的である．そのため超平面ハッシュ関数では，LOHA で考慮する必要のあったプロセッサ間の通信遅延の差をそれほど考慮しなくてもよいので，LOHA よりもずっと細かい粒度の仕事を制限されたプロセッサに割り当てることができ，そのような状況化での非同期計算は LOHA の場合とは大きくことなる．

9. 結　論

本論文では，HDA* を最適多重配列アラインメント問題に適用し，探索オーバーヘッドの増大による性能低下の原因を分析した．さらに，この問題の解決のために，超平面仕事分配法を提案し，東京工業大学の TSUBAME2.0 で 384 コアを利用した場合にも従来の HDA* よりもはるかに良い性能を得た．

今後の課題として，超平面ハッシュ関数の超平面の厚みをより適切に設定する方法の開発や，他のアプリケーションに超平面ハッシュ関数の考え方を適用することが挙げられる．現在の超平面ハッシュ関数は，多重配列アラインメントの空間の特徴に依存しているが，その考え方自体は自然なものであり，様々な空間でこのような性質が存在すると考えられる．応用分野の一例として，コストが均一でない古典的プランニングが挙げられる．このアプリケーションでは，多重配列アラインメント問題と同様に，本論文で示した HDA* の問題が生じると予想される．

参 考 文 献

1) P.E. Hart, N.J. Nilsson, and B.Raphael. 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, 1968.
2) A.Kishimoto, A.Fukunaga, and A.Botea. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'2009)*, pp. 201–208, 2009.
3) A.Kishimoto, A.Fukunaga, and A.Botea. On the Scaling Behavior of HDA*. In *Proceedings of the 3rd Symposium on Combinatorial Search (SoCS'2010)*, pp. 61–62. AAAI Press, 2010.
4) N.R. Mahapatra and S.Dutt. Scalable Global and Local Hashing Strategies for Duplicate Pruning in Parallel A* Graph Search. *IEEE Transactions on Parallel and Distributed Systems*, Vol.8, No.7, pp. 738–756, 1997.
5) M.Evett, J.Hendler, A.Mahanti, and D.Nau. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, Vol.25, No.2, pp. 133–143, 1995.
6) J.W. Romein, H.E. Bal, J.Schaeffer, and A.Plaat. A performance analysis of transposition-table-driven work scheduling in distributed search. *IEEE Transactions on Parallel and Distributed Systems*, pp. 447–459, 2002.
7) A.L. Zobrist. A new hashing method with application for game playing. Technical Report88, University of Wisconsin, 1970.
8) T.Ikeda and H.Imai. Enhanced A* algorithms for multiple alignments: Optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theoretical Computer Science*, Vol. 210, No.2, pp. 341–374, 1999.
9) J.D. Thompson, P.Koehl, R.Ripp, and O.Poch. BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark. *Proteins: Structure, Function, and Bioinformatics*, Vol.61, No.1, pp. 127–136, 2005.
10) R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, Vol.27, No.1, pp. 97–109, 1985.
11) R.E. Korf, W.Zhang, I.Thayer, and H.Hohwald. Frontier search. *Journal of the ACM (JACM)*, Vol.52, No.5, pp. 715–748, 2005.
12) R.Niewiadomski, J.N. Amaral, and R.C. Holte. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *Proceedings of the 21st National Conference on Artificial intelligence (AAAI'2006)*, pp. 1039–1044. AAAI Press, 2006.
13) R.E. Korf. Delayed duplicate detection: Extended abstract. In *International Joint Conference on Artificial Intelligence*, Vol.18, pp. 1539–1541. Citeseer, 2003.
14) E.Burns, S.Lemons, W.Ruml, and R.Zhou. Best-First Heuristic Search for multicore Machines. *Journal of Artificial Intelligence Research*, Vol.39, pp. 689–743, 2010.

ⓒ 2011 Information Processing Society of Japan