数理解析研究所講究録
第1676巻 2010年 51-65

大規模最短路問題に対する高速処理システム

- メモリ階層構造の考慮とクラスタ&クラウド技術による高速化 -

中央大学 理工学研究科経営システム工学専攻　安井 雄一郎 (Yuichiro Yasui)
Department of Industrial and Systems Engineering, Chuo University
NECシステムプラットフォーム研究所　高宮 安仁 (Yasuhito Takamiya)
System Platforms Research Laboratories, NEC Corporation
中央大学 理工学部　藤澤 克樹 (Katsuki Fujisawa)
Department of Industrial and Systems Engineering, Chuo University

概 要

最短路問題は経路探索などの多くの応用を持ち, また他の最適化問題の子問題として用いられることも多く, 適用範囲の広い組合せ最適化問題である. そのため, 最短路問題を高速に解くことは重要な意味を持つ. また, 最短路問題に対する解法には, 安定的かつ効率的な高速アルゴリズムが存在するが, 実問題は非常に大規模になるため, 高速化が不可欠である. メモリ階層構造を考慮し, 大規模最短路問題におけるダイクストラ法に対し計算機の理論演算性能を使い切るような高速化を行い, 本実装は先行研究と比べ, 実行性能, 安定性, メモリ要求量など総合的に最も優れたソルバを開発した. 実験により汎用的かつ客観的に評価を行い高速化の有用性を検証していく. さらに, 本ソルバを用いて開発した, 大規模最短路問題に対するオンライン・ソルバー, 経路探索高速処理システムについて説明を行う.

1. はじめに

最短路問題は最も基本的な組合せ最適化問題であり, 2006年に 9th DIMACS Implementation Challenge - Shortest Paths(9th DIMACS) [10, 11, 12] が開催されるなど, 現在も盛んに研究されている. 9th DIMACS では, 全米道路ネットワーク[1]を交差点を点, 交差点間の道路を枝とした非常に大規模なグラフ (約2400万点, 約5800万) を対象とし, 前処理には枝長が頻繁に変化しないという特性を利用し, 数十分から数時間の前処理を行うことにより, ユーザからの要求に高速に対応するよう設計されている. しかしながら, 渋滞情報・事故情報などを考慮した経路探索, 大規模災害時の避難経路探索などのリアルタイム性が非常に重要となる場合や, 他の最適化問題の子問題として実行される経路探索など, 前処理を行わずに高速な探索を要求される機会は少なくない. また, 前処理や前処理後の探索時にも, 前処理なしの経路探索を用いる実装は多く見られる.

非負の枝長を持つグラフに対する効率的なアルゴリズムとしてダイクストラ法 [1] が有名であり, アルゴリズム中の "次探索点の選択" が潜在的なボトルネックとなることが知られている. そこで, 探索候補点集合に優先キューと呼ばれるデータ構造を適用することで改善されてきた [2, 3, 4, 5, 6, 7, 8]. 中でもマルチレベル・バケット [8] は, RAM(Random Access Memory) モデル [9] を考慮されており, 理論的にも実験的にも高速な優先キューだが, メモリ要求量が大きく, マルチコアプロセッサ計算機の性能を十分に引き出すように設計されていない.

本研究では, バイナリ・ヒープを適用したダイクストラ法に対し, メモリ階層構造を考慮した高速化 [13, 14, 15] を適用することで, 実効性能, グラフ特性に対する安定性, メモリ要求量,

[1] http://www.census.gov/geo/www/tiger/

並列実行など、総合的に最も優れたソルバーを開発することを目的としている. 計算機上の律即箇所を解析するための実験方法を合わせて示し, 本研究の有用性を裏付ける.

さらに本実装を用いて開発した最短路問題オンラインソルバーについての説明を行い, クラスタ・コンピューティングとクラウド・コンピューティングとの融合による拡張である次世代経路探索処理システムにも言及する.

2. ダイクストラ法に対する優先キューの適用

2.1. 最短路問題に対するダイクストラ法

各枝に正の重みを持つ有向グラフ $G = (V, E)$ に対し, 2 種類の最短路問題について考える.

- 単一始点最短路問題 (Single-Source Shortest Path Problem; SS)
  - 入力 : 始点 s
  - 出力 : 始点から各点までの最短距離と最短路
- 一対一最短路問題 (Point-to-Point Shortest Path Problem; P2P)
  - 入力 : 始点 s, 終点 t
  - 出力 : 始点から終点までの最短距離と最短路

ダイクストラ法では各点 v に対し, 次のような作業領域が必要となる.

- $d(v)$ 始点からの距離を示すラベル
- $\pi(v)$ 直前に接続されている点
- $S(v)$ 探索状況を示すフラグ (`unreached`: 未探索, `labeled`: 探索済, `scanned`: 確定)

始点 s から探索する際には, $d(s) = 0$, $\pi(s) = \texttt{nil}$, $S(s) = \texttt{scanned}$, $d(v) = \infty$, $\pi(v) = \texttt{nil}$, $S(v) = \texttt{unreached}(v \in V \setminus s)$ と初期化し探索を開始する. 各反復時, 探索点 v の距離ラベル $d(v)$ と接続されている枝 $v - w$ の長さ (重み) $l(v, w)$ に, $d(v) + l(v, w) < d(w)$ が成立すれば $d(w) \leftarrow l(v, w) + d(v)$ と距離ラベルの更新を行い, $S(w)$ が `unrearched` であれば `labeled` とする. 探索点に接続されている枝をすべて探索し終わると $S(v) = \texttt{scanned}$ とし, `labeled` である距離ラベルが最小の点を新たに探索点として探索を繰り返す.

1 対 1 最短路問題では終点 t が探索点に, 1 対全最短路問題では全点のフラグが `scaned` になることがそれぞれ終了条件であり, グラフ中枝の重み負のサイクルが存在しなければ必ず正常に終了する. 各点は高々1 度ずつ探索点となり, 各枝は高々1 度ずつ探索される. ダイクストラ法の効率は次探索点決定の方法に依存するため, 探索点候補 (`labeled` である点集合) に対し適用した優先キューの性能・特性にアルゴリズムが依存する.

2.2. ダイクストラ法に対する優先キュー

最短路問題に対する優先キュー Q は, `insert`, `decreasekey`(`delete`, `insert`), `extract-min` という操作に対応したデータ構造であり, 大きく 2 種類 “ヒープを基とした優先キュー” と “バケツを基とした優先キュー” に分類が可能である.

- `insert`　点 $v \in V$ を, 距離ラベル $d(v)$ を優先度として優先キュー Q に挿入する.

- `delete`　点 $v \in Q$ を優先キューから削除する.
- `decrease-key`　点 $v \in Q$ を, 距離ラベル $d(v)$ を $d'(v)$ $(d'(v) < d(v))$ に更新する.
- `extract-min`　距離ラベル $d(v)$ が最小の点 $v \in Q$ を取り出す.

2.2.1. ヒープを基とした優先キュー

ヒープを基とした優先キューでは, 優先度の大小関係により木構造を構成するため, 扱うデータ値には性能が依存しにくく, 実行時間やメモリ要求量が安定的である. 最も基本的なバイナリ・ヒープ (Binary Heap)[2] は, 各親は 2 つの子を持ち, 親子間には「親の優先度 > 子の優先度」が常に成り立つ. 根に配置されているデータが最も優先度が高いように構成している. `insert`, `decrease-key`, `extract-min` の計算量はそれぞれ $O(log_2 n)$ となり, 同数のスワップ操作 (木構造を構成するためデータの入れ替え) が必要になる.

2.2.2. バケットを基とした優先キュー

バケットを基とした優先キューでは, 優先度の値に対し予め決定したルールに従いデータを配置させるため, 扱う値の取り方により実行時間やメモリ要求量が依存する. 最も基本的な 1 レベル・バケット (1-level buckets, Dial's algorithm)[3] は, 探索候補点集合の距離ラベルの取りうる値の範囲の幅が $C+1$(C:最大枝長) となることを利用して, $C+1$ 個のバケットから "$d(v)$ mod $C+1$" により 1 つを決定する. 同一のバケットに格納される点は距離ラベルが等しい. $O(1)$ である `insert` や `decrease-key` に対し, `extract-min` は $O(C+1)$ であるため, 最大枝長 C が大きくなるにつれ, 必要なバケツの数も増加し性能が低下する. また, 9th DIMACS でベンチマークとして使用されているマルチレベル・バケット (multi-level buckets; mbp)[8] は, 2 の累乗で分類するため, 優先度の値に性能が依存しにくい.

2.3. 優先キューの特性

バイナリ・ヒープ, 1 レベル・バケット, マルチレベル・バケットの 3 種類の優先キューを適用したダイクストラ法 2-heap, Buckets, mbp に対し, 点・枝の接続情報を固定し枝長 $l(e)$ を新たに $l'(e) = l(e) \times 2^t (^\forall e \in E)$ と, スケーリングした際の実行時間とメモリ要求量を図 1, 図 2 にまとめる. データに対し安定的な 2-heap や mbp に対し, Buckets は実行時間やメモリ要求量が最大枝長に依存しており, 道路ネットワーク ($t = 0$) とは比較的相性がよいが, 使用の際には注意しての決定すべきである.

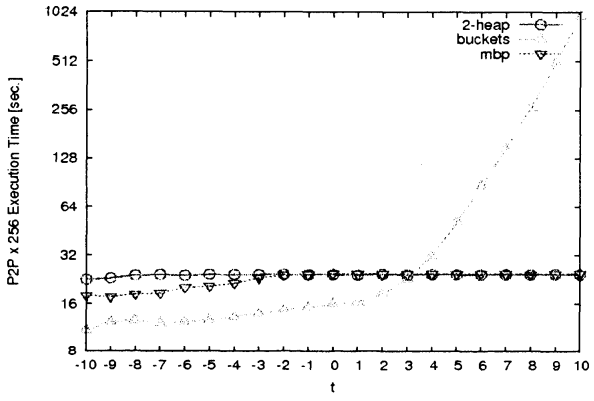


図 1: 枝長スケーリングによる実行時間の推移

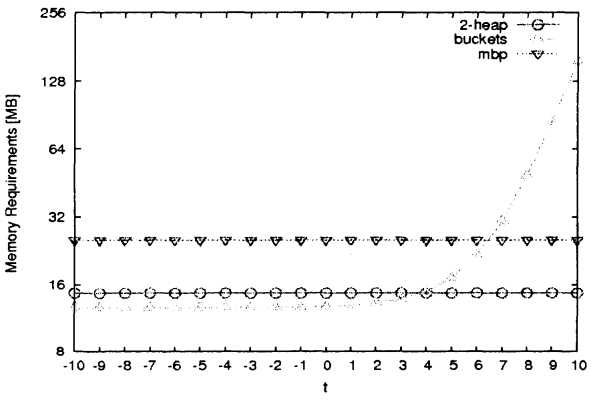


図 2: 枝長スケーリングによるメモリ要求量の推移

3. メモリ階層構造を考慮した高速化

3.1. 高速化の方針

メモリ階層構造を考慮することにより、特定のアーキテクチャや特定の問題特性に特化させることなく汎用的に高速化を行う. 特にマルチコア・プロセッサ計算機環境を想定し、多くのコア上で実行するためにメモリ要求量を抑えるように設計する. C プログラミング言語で開発する.

性能効率を出しやすい条件としては、"データ移動量に対し計算量がある程度大きいこと" や、"データアクセスが連続的で、中長期的に予測が可能であること" が挙げられるが、大規模最短路問題におけるダイクストラ法は、"データ移動量に対し演算量が非常に小さく"、"不連続な領域に対しデータアクセスが広域に及び、中長期的な予測が困難" と容易ではないことに注意されたい.

3.2. メモリ階層構造

計算機を図 3 のようなメモリ階層構造で捉える [16]. メモリ階層構造では、上位レベルになるほど高速で小容量な記憶容量を、下位レベルになるほど低速だが大容量な記憶容量を保持している. 高速化において、演算量とデータ移動量の割合を考慮し適切に整え、より上位の高速なキャッシュメモリを可能な限り利用することは非常に重要である.

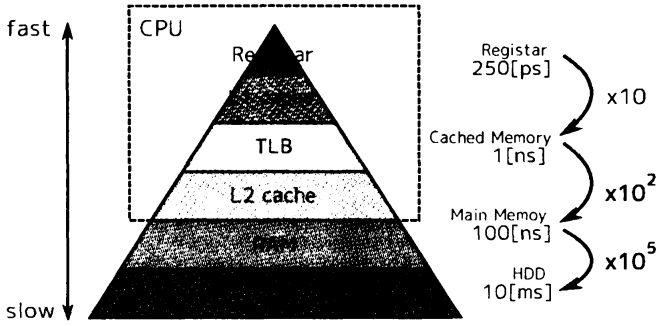


図 3: メモリ階層構造

3.3. データアクセスパターンを考慮したデータ配置

データアクセスパターンの中長期予測が可能ならば、必要となるデータを高速なキャッシュメモリに予め配置しておくことでメモリ移動コストを隠蔽することが可能になるものの、ダイクストラ法では動的計画法の特性から中長期的な予測が困難であり、データの再利用率も低い. そこで局所的にアクセスパターンを考慮してデータ配置を整えることにより、データアクセスの密度を改善する. グラフ表現に距離行列を用いたダイクストラ法は、図 4 のように広域に渡り不連続なアクセスを繰り返してしまうが、フォワードスター・グラフ表現 (図 6) を適用し同一始点に接続される枝情報を連続的に配置させると図 5 のように、局所的に連続的なデータアクセスパターンとなる.

ダイクストラ法に対するフォワードスター・グラフ表現では、点数分の始点に相当する枝情報へのポインタ (図 7) と、枝数分の終点と枝の重みを持つ枝情報 (図 8) で表現されるが、それ

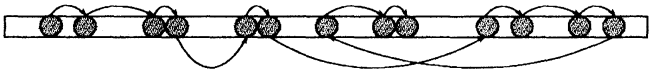


図 4: 不連続なデータアクセスパターン

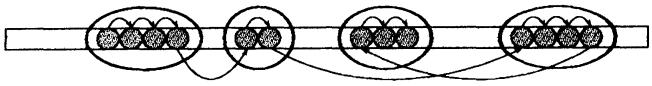


図 5: 局所的に連続なデータアクセスパターン

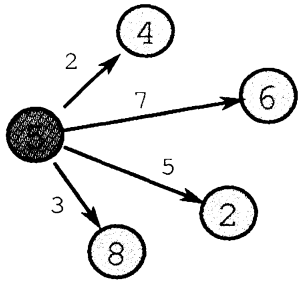


図 6: フォワードスター・グラフ表現

index	arc-index
4	-
5	**14**
6	**18**
7	-

図 7: 枝情報へのポインタ

index	head	length
13	-	-
14	4	2
15	6	7
16	2	5
17	8	3
18	-	-
19	-	-

図 8: 枝情報

ぞれ関連したデータアクセスが行われるため, 配置方法 1 のよりも**配置方法 2** のような配置により, データアクセスの密度が改善される. 各点における作業領域 (距離ラベル, 直前点 ID, 優先キューへの逆引き) に対しても同様である.

```
int head[NUM_ARCS], lengh[NUM_ARCS];
```

図 9: **配置方法 1**: データを個別に連続的に配置

```
struct arc_t {
  int head, length;
} arc[NUM_ARCS];
```

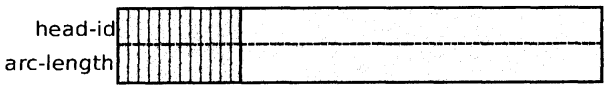


図 10: **配置方法 2**: 要素をまとめて連続的に配置

現在の一般的なプロセッサのキャッシュ・アライメントは 64 バイトであるため, 1 度のデータアクセスで 64 バイト (32 ビット整数型 (4 バイト) であれば 16 要素) のデータをロードすることが可能である. 全米道路ネットワークグラフの各点から接続している枝は高々数本であるため, 多くの場合 1 度のデータアクセスで必要な枝情報を得ることが可能になる. 図 11, 表 1 は, "グラフ表現における枝情報" と" 各点における作業領域" に対して, 配置方法による性能差をまとめたものである. $a1, a2, n1, n2$ は, それぞれグラフ表現における枝情報, 各点における作業領域に対し, 配置方法 1, 配置方法 2 の採用を表わしている. グラフ表現, 各点の作業領域ともに配置方法 1 で実装したダイクストラ法 $(a1, n1)$ を基準 (100%) としている. 各計算機環境毎 (表 14) の性能特性はあるものの, 性能順は一致している.

$$(a1, n1) < (a2, n1) < (a1, n2) < (a2, n2)$$

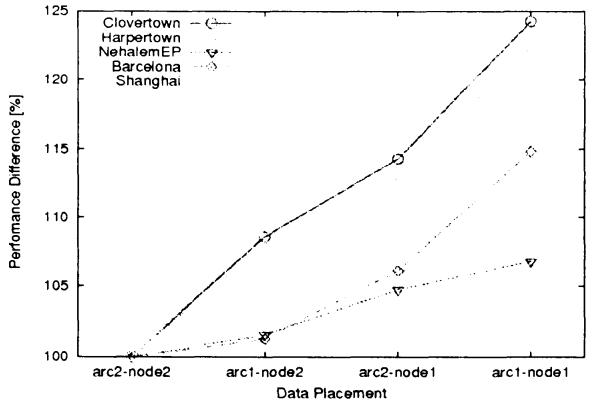


図 11: メモリ配置による性能差 [%]

表 1: メモリ配置による性能差 [%]

	a2-n2	a1-n2	a2-n1	a1-n1
Clovertown	-	+8.58%	+14.29%	+24.25%
Harpertown	-	+8.04%	+12.94%	+22.35%
NehalemEP	-	+1.57%	+ 4.76%	+ 6.79%
Barcelona	-	+1.28%	+ 6.11%	+14.83%
Shanghai	-	+9.09%	+15.49%	+23.36%

3.4. バイナリ・ヒープにおけるスワップ操作の改善

配列で実装されたバイナリ・ヒープはデータの空間的局所性が非常に高い優先キューであるが、ヒープ構造を保持するための不連続なデータアクセスとスワップ操作により、計算量から期待される性能を引き出すことは容易ではない. 特にスワップ操作は、" 書込み直後に読込み" を行うため、ライトバック形式キャッシュメモリの特性を無効化してしまう. バイナリ・ヒープでは計算順序入れ換えにより、スワップ操作を排除することが可能である. スワップ操作を用いたものを `topdown-extract-min`(図 12, Algorithm1)、スワップ操作を排除したものを `bottomup-extract-min`(図 13, Algorithm2) とする.

表 2 は、`topdown-extract-min` と `bottomup-extract-min` を用いたバイナリ・ヒープを適用したダイクストラ法の実行時間をまとめたものである. 全米道路ネットワークグラフに対し、1 対 1 最短路問題を繰り返し計算した際の平均の実行時間であるが、実行時間自体は倍ほど差はあるもののいずれも 4, 5%程と同率の改善が確認できる.

表 2: スワップ排除による性能向上の割合 [%]

計算機環境	`topdown-extract-min` [秒/クエリ]	`bottomup-extract-min` [秒/クエリ]	性能向上
Clovertown	3.604	3.405	+5.52%
Harpertown	2.490	2.371	+4.78%
Nehalem-EP	2.561	2.413	+5.78%
Barcelona	4.474	4.301	+3.87%
Shanghai	3.757	3.576	+4.82%

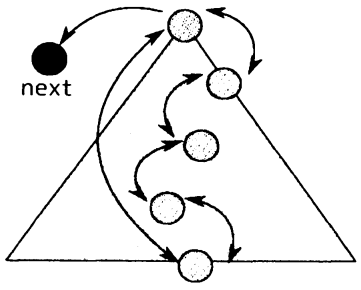


図 12: `topdown-extract-min`

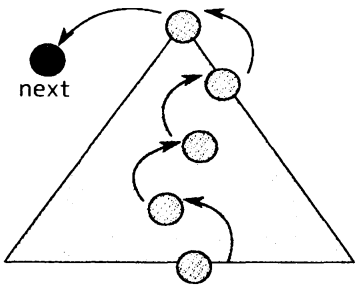


図 13: `bottomup-extract-min`

Algorithm 1 `topdown-extract-min`

```
1: if H = ∅ then
2:    return nil
3: end if
4: x ← H[root], H[root] ← H[H.size], i ← root
5: while H[left[i]] ≠ ∅ do
6:    if H[left[i]].key < H[right[i]].key or H[right[i]] = ∅ then
7:       min ← left[i]
8:    else
9:       min ← right[i]
10:   end if
11:   if H[min].key < H[i].key then
12:      swap(H[i], H[min])
13:      i ← min
14:   else
15:      return x
16:   end if
17: end while
18: return x
```

Algorithm 2 `bottomup-extract-min`

```
1: if H = empty then
2:    return nil
3: end if
4: x ← H[root], i ← root
5: while H[left[i]] ≠ ∅ do
6:    if H[left[i]].key < H[right[i]].key or H[right[i]] = ∅ then
7:       min ← left[i]
8:    else
9:       min ← right[i]
10:   end if
11:   if H[min].key < H[i].key then
12:      H[i] ← H[min]
13:      i ← min
14:   else
15:      break
16:   end if
17: end while
18: H[i] ← H[tail]
19: return x
```

3.5. ダイクストラ法のクエリ並列化

ダイクストラ法は各反復における依存関係が非常に強く、大規模なグラフに対しても数秒で終了するため、並列実行向きのアルゴリズムとはいえない。アルゴリズム内の並列化による性能向上は期待することは困難であるため、複数のクエリに対し複数のダイクストラ法を並列計算させることで、マルチコア・プロセッサ計算機環境の性能を引き出すことが可能である。POSIX Pthread ライブラリを用いる。

4. メモリ階層構造上のボトルネック箇所の解析

メモリ階層構造を考慮し、汎用的かつ客観的な評価を行うための実験方法を示す。プロファイラによる律速箇所の限定は非常に有効であるが、微小区間の計測などプロファイラ自身のオーバーヘッドにより正しく測定できない場合も少なくなく、また、詳細なプロファイル結果を得られても計算機のどの箇所に律速しているか判断することは困難である。そこで計算機実験により、メモリ階層構造上のどの箇所にどのように律即するか把握を行う。以下の実験結果は複数の計算機環境 (表 14) に対し、高速化後のバイナリ・ヒープを適用したダイクストラ法を用いて行ったものである。

4.1. プロセッサの動作周波数を変化させて実行

メモリ帯域幅は固定しプロセッサの動作周波数のみを変化させる。その際の実行時間の変化によりメモリ帯域幅に律速されている否かを判断する。`cpufreq-selector` コマンドを用いる。表 3 のように、メモリ帯域幅に律速しているのであれば、動作周波数を変化させても実行時間は変化しない。

表 3: プロセッサの動作周波数変化実験によるボトルネック推定箇所

実行時間	ボトルネック箇所
性能変化なし	メモリ帯域幅に律速されている
性能変化あり	メモリ帯域幅以外に律速されている可能性が非常に高い

Intel Xeon 5160 の本来 3.00GHz であった動作周波数を 2.00GHz (-33%) まで低下させると実行時間は 37% 長くなるため、メモリ帯域幅以外に主要因が存在すると判断される。

表 4: プロセッサの動作周波数変化による実行時間の変化量 [%]

	元の動作周波数	変化後の周波数 (変化量)	実行時間の変化量
Xeon 5160	3.00GHz	2.00GHz(-33 %)	**+37%**

4.2. 2 プロセッサ・コア同時実行

1 プロセッサ・コア上での実行時間と特定の 2 プロセッサ・コア上で同時実行した際の実行時間を比較することで、ボトルネック箇所を特定する。クアッドコア・プロセッサを 2 基搭載した計算機環境では、コア指定の組合せは表 5 となる。`numactl` コマンドを用いる。ボトルネックの限定箇所と条件をまとめたものが表 6 である。

実験結果から、律速箇所はメモリ帯域幅ではなく、L2 キャッシュメモリの共有によるものだと確認される (表 7)。つまり、スレッド並列時に L2 キャッシュメモリを共有しないようにコアを割り振れば、並列数分だけ性能を得ることが可能である。

表 5: クアッドコア・プロセッサにおける 2 コアの組合せ

コアの組合せ	詳細
1 コア	基準とする実行時間
異なるソケット	異なるソケット上の 2 コア
非共有 L2 キャッシュメモリ	同一ソケット上の L2 キュッシュメモリを共有しない 2 コア
共有 L2 キャッシュメモリ	同一ソケット上の L2 キュッシュメモリを共有する 2 コア

表 6: 2 プロセス同時実行によるボトルネック箇所の限定

律速している箇所	他ソケット	非共有 L2	共有 L2
メモリ帯域幅 (1 プロセス分)	変化あり	変化あり	変化あり
メモリ帯域幅 (2 プロセス分)	変化なし	変化あり	変化あり
L2 キャッシュメモリの共有	変化なし	変化なし	変化あり
演算性能	変化なし	変化なし	変化なし

表 7: 2 プロセッサコア同時実行での性能低下率 [%]

	異なるソケット	L2 非共有 2 コア	L2 共有 2 コア
Clovertown	-0.62%	+3.47%	**+25.20%**
Harpertown	+1.10%	+3.58%	**+21.13%**
Nehalem-EP	-0.16%	+3.89%	-
Barcelona	+1.16%	+5.28%	-
Shanghai	-0.47%	+2.24%	-

4.3. 2 プロセッサ・コア同時読込み

L2 キャッシュメモリを共有することによる性能低下の要因に関して詳細を解析していく. 同一領域に対し 2 プロセッサ・コアで同時にデータ参照を行った際のメモリ帯域幅を測定することで, "読込みのみ" であるか "書込みを伴う読込み" であるか判定することが可能である. Intel Xeon X5460 3.16GHz × 2 上で参照するデータサイズを変化させながら実行すると, 図 14 が得られる.

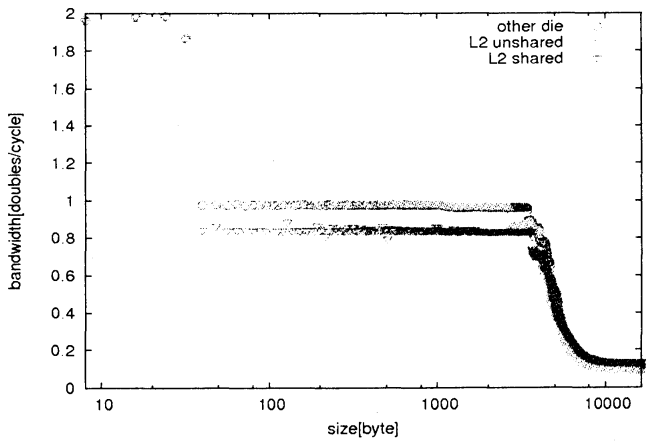


図 14: 2 プロセッサ・コア同時読込みにおけるメモリ帯域幅 [doubles/cycle]

L2 キャッシュメモリを参照しているデータサイズ (数十キロバイトから 6 メガバイト) では, 他の場合に比べ 16 % ほど低下している. 2 プロセッサ・コア同時実行による結果である 21 % と比較すると, L2 キャッシュメモリ共有によるレイテンシが主要因であることが確認される.

5. メモリ階層構造による高速化後の性能

全米道路ネットワークグラフ (表 8) を用いて, バイナリ・ヒープ, 1 レベル・バケット, マルチレベル・バケットを適用したダイクストラ法 (表 9) に対して性能比較実験を行う.

表 8: 全米道路ネットワークグラフ

変数名		詳細
点数	23,947,347	交差点
枝数	58,333,344	交差点間の道路
枝長	[1, 368855]	枝長の範囲
次数	[1, 8]	点に接続している枝の数

表 9: 比較する優先キュー

	1 対全最短路問題の計算量	優先キュー
2-heap	$O(m \log_2 n)$	高速化後のバイナリ・ヒープ (領域を静的確保)
2-heap*	$O(m \log_2 n)$	高速化後のバイナリ・ヒープ (領域を動的確保)
Buckets	$O(m + nC)$	高速化後の 1 レベル・バケット
mbp	$O(m + n \log C)$	マルチレベル・バケット

5.1. 全米グラフデータに対するバイナリ・ヒープを適用したダイクストラ法

表 10, 図 15 は, 2-heap の計算機環境毎 (表 14) の 1 対 1 最短路問題クエリ毎の実行時間をまとめたものである ([スレッド並列数]). L2 キャッシュメモリを 2 プロセッサ・コアで共有するクアッドコア・プロセッサ (Clovertown, Harpertown) の 8 スレッド並列時の性能は, 4 スレッド並列までと比べて効率が低下しているが, プロセッサ・コア毎に L2 キャッシュメモリを保持しているクアッドコア・プロセッサ (NehalemEP, Barcelona, Shanghai) では性能低下なしにスレッド並列数分の台数効果を得られることが確認される.

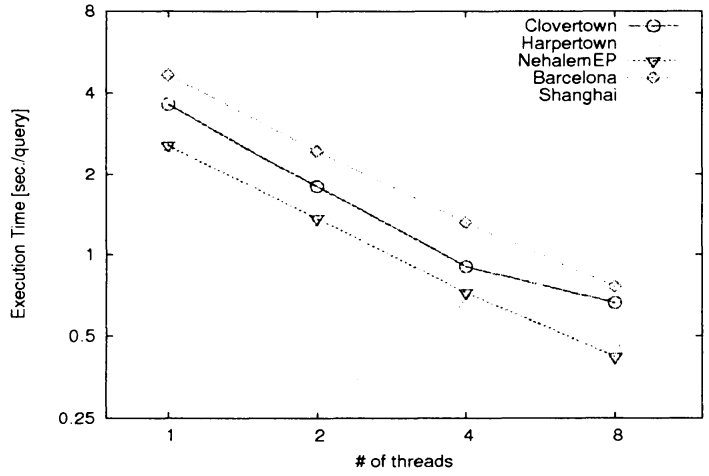


図 15: ダイクストラ法の実行時間 [秒/クエリ]

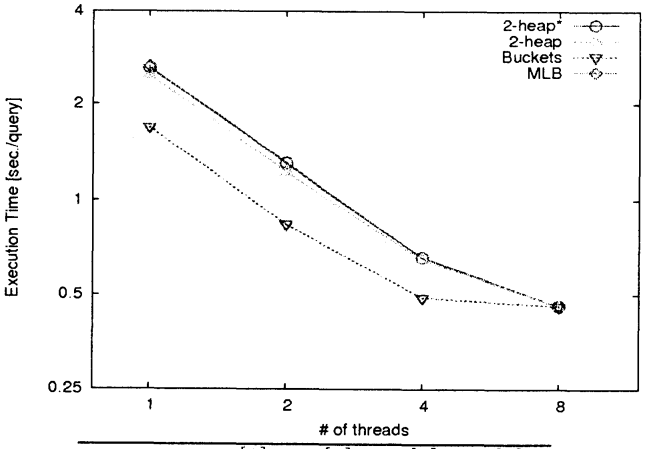
表 10: ダイクストラ法の実行時間 [秒/クエリ]

	[1]	[2]	[4]	[8]
Clovertown	3.64	1.80	0.90	0.66
Harpertown	2.47	1.22	0.65	0.47
Nehalem-EP	2.58	1.36	0.72	0.42
Barcelona	4.67	2.46	1.31	0.76
Shanghai	3.86	2.01	1.03	0.57

5.2. 1対1最短路問題に対する優先キュー毎のダイクストラ法の性能

メモリ階層構造を考慮し実装を高速化を行った 2-heap, 2-heap*, Buckets は, mbp と比べいずれも省メモリで非常に効率的な台数効果を得られている (図 11,12, 表 11,12). 特に 2-heap* はマルチコア・プロセッサ計算機環境において, 非常に少ないメモリ要求量で実行時間がグラフデータに依存度の少ない安定的な実行が可能である. mbp と比べ同量のメモリ要求量で 4 倍程 (4 スレッド並列) の性能を示している.

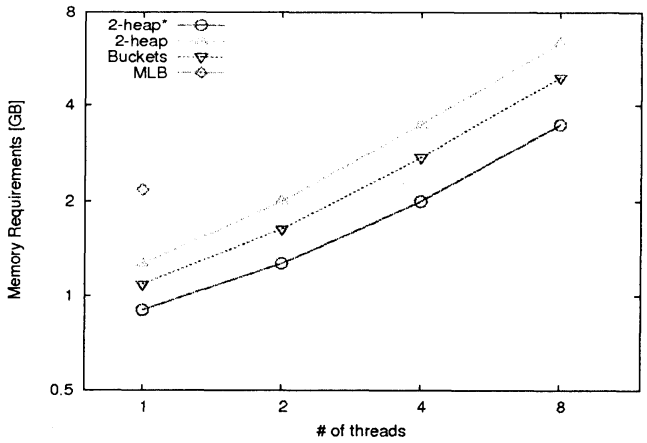
図 16: 優先キュー毎の実行時間 [秒/クエリ]



	[1]	[2]	[4]	[8]
2-heap*	2.61	1.30	0.66	0.46
2-heap	2.47	1.22	0.65	0.46
Buckets	1.68	0.84	0.49	0.46
mbp	2.65	-	-	-

表 11: 優先キュー毎の実行時間 [秒/クエリ]

図 17: 優先キュー毎のメモリ要求量 [GB]



	[1]	[2]	[4]	[8]
2-heap*	0.90	1.27	2.00	3.46
2-heap	1.27	2.00	3.46	6.38
Buckets	1.09	1.64	2.73	4.93
mbp	2.17	-	-	-

表 12: 優先キュー毎のメモリ要求量 [GB]

6. 高速経路探索処理システム

6.1. 最短路問題オンラインソルバー

我々は, 9th DIMACS で公開されている道路ネットワークグラフを用いた大規模最短路問題を, ウェブブラウザによる GUI(グラフィカル・ユーザ・インターフェイス) で利用可能な最短路問題オンラインソルバー[2]を公開している (図 18). 本システムで用いている最短路問題ソルバーは, 本研究で開発した前処理行わない厳密解を計算するダイクストラ法である. グラフデータを予めメモリ上に配置しているのではないため, ネットワークと表示にかかるオーバーヘッドを除けば, 本研究で開発した最短路問題ソルバの性能を体験することが可能である.

6.1.1. 最短路問題オンライン・ソルバーのシステム概要

本システムは図 19 のように構成されている. ユーザは, 一般的なブラウザからアクセスし, マウスなどのポインティング・デバイスを用いて, 始点や終点 (経路点を選ぶことが可能である) を指定する. ユーザからクエリを受け付けたサーバに指定された最短路問題を計算し, 結果を経路ファイルもしくは画像ファイルを出力し, 表示する. 特に最短路問題ソルバーによる計

[2]http://opt.indsys.chuo-u.ac.jp/portal/

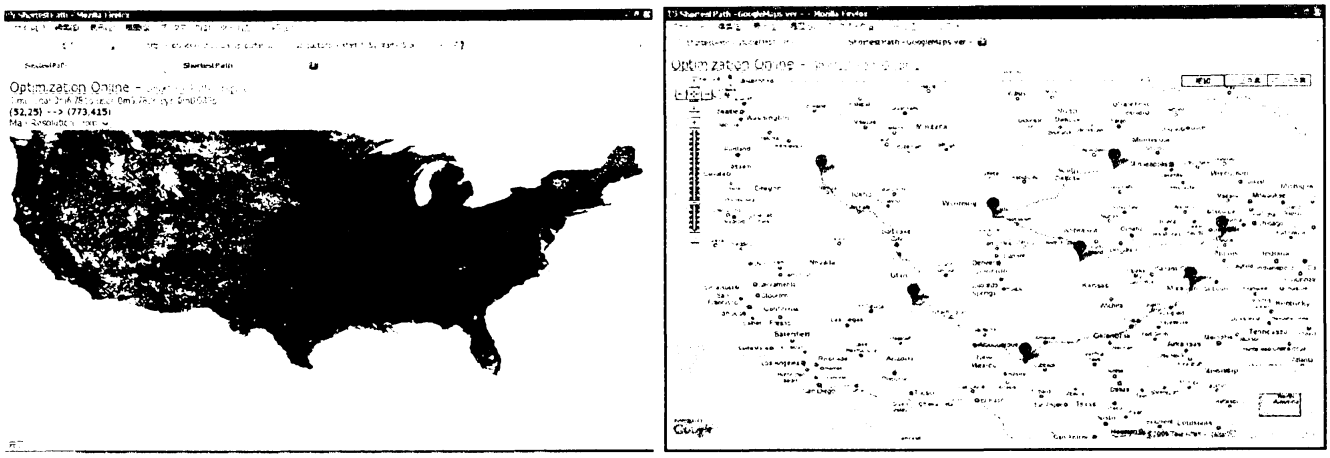


図 18: 最短路問題オンラインソルバーの実行画面

算部分は、フロントエンドサーバでなくてもよく、クラスタ・コンピューティングやクラウド・コンピューティングとの連携により、より大規模な問題に対応することが可能になる.

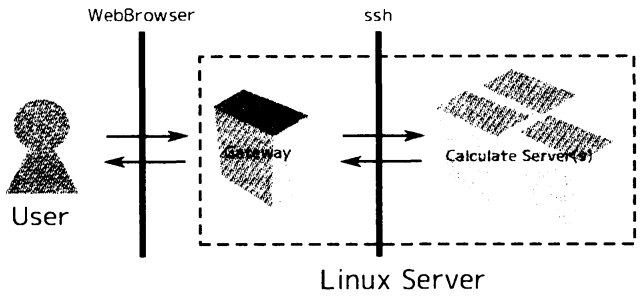


図 19: 最短路問題オンライン・ソルバーのシステム概要

7. 次世代経路探索処理システム

7.1. 次世代ナビゲーションシステム

現在のカー・ナビゲーションシステムでは、小さな機器上で経路探索を行う必要があるため、ユーザからの要求に対し高速で応答できるよう何らかの前処理により高速化を行っている. その弊害として、出力結果が明らかに最短路でないことも珍しくなく、渋滞情報や事故情報を考慮して経路探索を行うことが非常に困難であるため、前処理なしで高速に経路探索処理を行うシステムが必要である.

システムに対して常に同量の要求が来るとは限らず、日中に対し夜間は少なくなることが予想される. システム要件を満たす計算機資源を決定することは非常に困難であるが、クラウド・コンピューティングによる計算資源の動的追加/削除による柔軟な対応により、それに伴ない、不要な管理コストや使用電力などを抑えることが可能である (図 20)..

7.1.1. Lucie EC2

Lucie EC2 は、Linux クラスタ・コンピュータ用の自動管理ツール Lucie のクラウド・コンピューティングに対する拡張である. Lucie EC2 を用いることで、予め確保する計算機資源

(private cloud) とクラウド・コンピューティングにより動的に追加された計算機資源 (public cloud) を統合し, 抽象化した計算機資源 (virtual cloud) として扱うことが自動化される (図 20).

7.1.2. スケジューラの方針

計算機へのクエリの割り振りと計算機資源の増減の方針をスケジューラとして実装している. 現時点では次のように実装しているが, 最適化問題に帰着しより効率化を図ること予定している. また, 稼働中の計算機全体を考慮し, 複数の性能の劣る計算機を性能の良い計算機にまとめることで効率化を図ることができるだろう.

- 計算機資源が不足時 → すぐに追加
- 課金単位時間が終了時 → 不必要のであると判断した場合停止

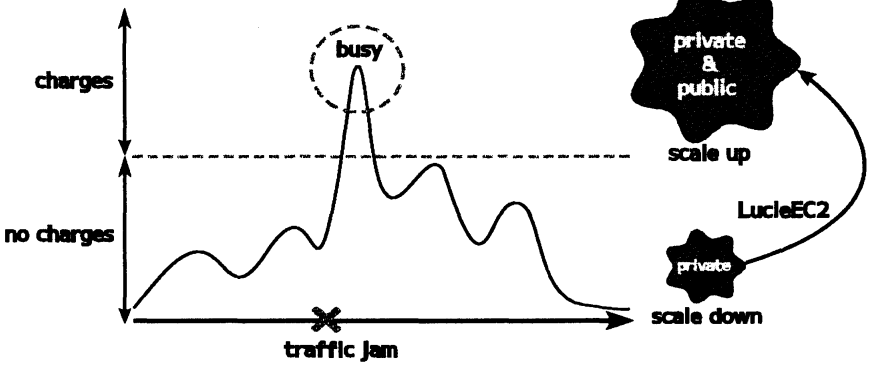


図 20: LucieEC2 による計算機資源の動的追加&削除

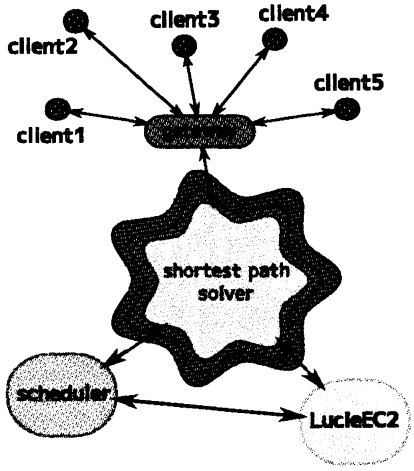


図 21: 高速経路探索処理システム概要

8. おわりに

本研究ではメモリ階層構造を考慮することにより, 大規模最短路問題に対するダイクストラ法に対して非常に汎用的かつ強力な高速化を行うことが可能であることを示した. 本研究で高速化を施したバイナリ・ヒープを適用したダイクストラ法は, メモリ要求量を抑えながらマルチコア・プロセッサ計算機性能を引き出すことが可能となり, 先行研究のマルチレベル・バケットに対して 1 スレッド時には同程度の性能を示し, メモリ要求量が同量となる 4 スレッド時には 4 倍ほども高速である. また, 数値実験による律速箇所の限定や解析により, バイナリ・ヒープを適用したダイクストラ法は, L2 キャッシュメモリのレイテンシに律速していることが判明し, L2 キャッシュメモリを共有しないプロセッサコアの組合せでは, 非常に効率的なマルチスレッド計算が可能である. グラフデータに対するメモリ要求量や実行時間の安定性, 省メモリ性, マルチコアプロセッサ環境での性能など, 総合的に評価すると最も優れているといえる. 本研究で扱った実装方法, 評価手法は非常に汎用的であり, ダイクストラ法や優先キューだけに限定せずにアルゴリズム実装に広く適用可能で高速化に期待ができる.

また, 最短路問題オンライン・ソルバーとしてウェブ上に公開しているため, ウェブブラウザによる GUI 操作で容易にソルバーの性能を確認可能である. 非常大規模な実道路ネットワークグラフ (約 2400 万点, 約 5800 万枝) に対し, リアルタイムかつ高速計算 (数秒) しているシステムは類を見ない. さらに現在開発中の大規模最短路問題に対する高速処理システムの概要と

今後の計画について説明を行った．本システムは，Lucie EC2 を用いてクラウド・コンピューティングを計算機資源の自動増減を行うことが可能である．

9. 謝辞

テキサス州立大学の後藤和茂氏には，メモリ階層構造を考慮しての高速化について，多くの貴重なご助言を頂きました．ここに感謝の意を表します．

A. 計算機環境

表 13: コンパイラ

コンパイラ	gcc-4.1.2, gcc-4.4.2
最適化オプション	-O2

表 14: 計算機環境

	プロセッサ	動作周波数	搭載メモリ	GCC
Clovertown	Intel Xeon(R) E5345 × 2 (4cores×2)	2.33 GHz	16 GB	4.1.2
Harpertown	Intel Xeon(R) X5460 × 2 (4cores×2)	3.16 GHz	48 GB	4.4.2
Nehalem-EP	Intel Xeon(R) X5550 × 2 (4cores×2)	2.67 GHz	72 GB	4.4.2
Barcelona	AMD Opteron(tm) 2356 × 2 (4cores×2)	2.30 GHz	36 GB	4.4.2
Shanghai	AMD Opteron(tm) 2384 × 2 (4cores×2)	2.70 GHz	36 GB	4.1.2

参考文献

[1] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1:269.271. 1959.

[2] J.W. J.Williams. Heapsort. Communications of the ACM, 7:347.348. 1964.

[3] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. Comm. ACM, 12:632-633 1969.

[4] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. J. Assoc. Comput. Mach., 34:596-615. 1987.

[5] Andrew V. Goldberg, and C. Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. Technical Report 95-187, NEC Research Institute, Inc. 1995.

[6] B. V. Cherkassky, Andrew V. Goldberg, T Radzik. Shortest paths algorithms: theory and experimental evaluation. Mathematical programming. 1996.

[7] B. V. Cherkassky, Andrew V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. SIAM J. Comput., 28:1326-1346. 1999.

[8] Andrew V. Goldberg. A Simple Shortest Path Algorithm with Linear Average Time. Technical Report STAR-TR-01-03, STAR Lab., InterTrust Tech., Inc., Santa Clara, CA, USA. 2001.

[9] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley. 1974.

[10] 9th DIMACS Implementation Challenge. `http://www.dis.uniroma1.it/~challenge9/`.

[11] Camil Demetrescu, Andrew V. Goldberg, David S. Johnson. 9th DIMACS Implementation Challenge Core Problem Families. 2006.

[12] Camil Demetrescu, Andrew V. Goldberg, David S. Johnson. 9th DIMACS Implementation Challenge Benchmark Solvers. 2006.

[13] GotoBLAS. `http://www.tacc.utexas.edu/resources/software/`.

[14] Kazushige Goto, K. and Robert A. van de Geijn. On reducing TLB misses in matrix multiplication. Tech.Rep. CS-TR-02-55. 2002.

[15] Kazushige Goto, K. and Robert A. van de Geijn. High-performance implementation of the level-3 BLAS. FLAME Working Note #20 TR-2006-23. 2006.

[16] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach (Third Edition ed.). Morgan Kaufmann Publishers.

[17] Lucie EC2. `http://lucie.is.titech.ac.jp/trac/lucie/`