Fast PGAS Implementation of Distributed Graph Algorithms

秋山 茂樹

全体ミーティング (2011/1/26)

Fast PGAS Implementation of Distributed Graph Algorithms

- PGAS 言語 UPC を用いて、分散環境における高速な並列グラフアルゴリズムを実装
  - グラフの連結成分を求めるアルゴリズム
  - 最小全域木を求めるアルゴリズム
- SMP 実装をベースとして、局所性を考慮して分散メモリアクセスを最適化

Fast PGAS Implementation of Distributed Graph Algorithms

- 著者
  - Guojing Cong,
    George Almasi, Vijay Saraswat
    (IBM TJ Watson Research Center)
- 発表場所
  - SC10 -
    the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis

発表の流れ

1. 導入
2. 共有メモリ実装
3. PGAS 実装
4. 局所性を用いた高速化
5. いくつかの改善
6. まとめ

1. 導入

Large-scale graph analysis

- 大規模グラフ解析は挑戦的課題
  - Memory-intensive
  - 不規則なメモリアクセス
- 効率の良いSMP実装は研究されてきているが分散メモリ実装はあまり例がない
  - 不規則なアクセスが効率低下の主因
  - 成功例: BFS on BlueGene/L

1. 導入

PGAS languages

- メモリ空間をスレッドごとに分割
  - 共有メモリ的考え方で並列プログラミング可能
    - 簡単(と言われる)
    - チューニングしやすい(RDMAがそのまま使える)

- 共有メモリでのアルゴリズムをそのまま利用可
  - そのままではなかなか性能は出ない...
    - スレッド間通信のレイテンシ大
      => 不規則なアプリだと不利

1. 導入

PGAS language contstructs

- 共有メモリと同様のコンストラクト (UPC)
  - グローバルメモリ割り当て
  - グローバルメモリに対する memcpy
  - 集約通信: gather, broadcast, reduce, ...
  - バリア, ロック
  - Work sharing: upc_forall
  - etc...

Contributions

- PGAS上でSMP向け並列グラフアルゴリズムを実装
  - 連結成分(CC) と 最小全域木(MST)
  - 単純な実装の性能評価
  - **通信効率/キャッシュ性能を高める手法の提案**
    - 不規則なメモリアクセスのまとめ上げ
  - その他
    - アルゴリズムレベルの改善
    - PGASに特化したいくつかの改善

2. SMP 実装

連結成分 (CC)

- グラフ G = (V, E) の連結成分を計算
  - Shiloach-Vishkin アルゴリズム

2. SMP 実装

SMP上でのCC実装

初期状態: D[i] = i (0 <= i <= n - 1)

```
for (;;) {
  gr = 0;
  parallel for i = 0 to m {
    (u,v) = E[i];
    if (D[u] < D[v]) {
      D[D[v]] = D[u];
      gr = 1;
    }
  }
  gr = node_Reduce_i(gr, SUM);
  if (gr == 0) break;
  parallel for i to n {
    while (D[i] != D[D[i]])
      D[i] = D[D[i]];
  }
}
```

Grafting:
各辺について、
両端の頂点に同じ連結成分番号を与える

終了条件:
連結成分番号に変化が生じなくなれば終了

Short-cutting:
各頂点について、
同じ番号を持つ頂点の頂点番号が変わっていたら、その番号に書き換える

3. PGAS 実装

Naiveな実装

- UPCを用いてSMP実装とほぼ同じ形で実装

UPC

```
for (;;) {
  gr = 0;
  upc_forall (i=0;i<m;i++) {
    u = E[i].u; v = E[i].v;
    if (D[u] < D[v]) {
      D[D[v]] = D[u];
      gr = 1;
    }
  }
  gr = all_reduce_i(gr, SUM);
  if (gr == 0) break;
  upc_forall (i=0;i<n;i++) {
    while (D[i] != D[D[i]])
      D[i] = D[D[i]];
  }
}
```

SMP

```
for (;;) {
  gr = 0;
  parallel for i = 0 to m {
    (u,v) = E[i];
    if (D[u] < D[v]) {
      D[D[v]] = D[u];
      gr = 1;
    }
  }
  gr = node_Reduce_i(gr, SUM);
  if (gr == 0) break;
  parallel for i to n {
    while (D[i] != D[D[i]])
      D[i] = D[D[i]];
  }
}
```

3. PGAS 実装

評価

- SMP (16 threads) vs. UPC (256 threads)

3. PGAS 実装

なぜ遅いか

- 分散メモリへのアクセスが不規則であり、細粒度の通信が多数発生するため
  - SMPでも問題になるが、分散環境ではメモリアクセスのレイテンシがより大きい

```
for (;;) {
  gr = 0;
  upc_forall (i=0;i<m;i++) {
    u = E[i].u; v = E[i].v;
    if (D[u] < D[v]) {
      D[D[v]] = D[u];
      gr = 1;
    }
  }
  ...
```

それぞれ通信が発生

SMPクラスタで性能を出すためには

- メモリ階層を意識し、通信効率と
  キャッシュ性能を最適化してやる必要がある
  - (特に)メモリアクセスの**空間的局所性**を改善すべき

4. 局所性を用いた高速化

通信効率の改善

- 複数の通信を一つの通信にまとめる
  (communication coalescing)
  - Collective (GetD) を用意:
    - input: アクセスしたい要素の index のリスト
    - output: アクセスしたい要素のリスト
    - **集合通信**を用いて index をノード間で交換し、
      同じノードへの通信をまとめて送信(put)

※ GetDと同じようにしてSetDも実装可

4. 局所性を用いた高速化

GetD

Input: **shared** array D, **private** array $indices$ with global indices to D, size of $indices$ is k

Output: **private** output array $outD$

begin

1. Sort $indices$ using the target thread id of $D[indices[l]]$ $(0 \leq l \leq k-1)$ as key
2. Compute number of elements to request from each thread $needFrom[j]$ $(0 \leq j \leq p-1)$
3. Inform all threads of number of elements and their target locations to sent to thr_i

$s \leftarrow 0$

for $j \leftarrow 0$ *to* $p-1$ **do**

3.1 $SMatrix[j][i] \leftarrow needFrom[j]$

3.2 $PMatrix[j][i] \leftarrow s$

3.3 $s \leftarrow s + needFrom[j]$

end

4. All threads enter barrier

UPC_barrier;

5. Inspect received requests and transfer data elements in batches

for $j \leftarrow 0$ *to* $p-1$ **do**

5.1 $nSend \leftarrow SMatrix[i][j]$

5.2 $targetPosition \leftarrow PMatrix[i][j]$

5.3 Collect data indices requested by thread thr_j

5.4 Gather the data elements

5.5 Put data to thr_j at position $targetPosition$

end

6. Permute received data to match the request order

end

Algorithm 2: *GetD* on thread thr_i, for $(0 \leq i \leq p-1)$.

各ノードに対して
データの要求 (indices)
を書き込む: O(p)

各ノードからの要求を
読んで、それぞれ指定
された位置にデータを
書き込む: O(p)

GetDの通信イメージ (1/2)

- インデックスの交換
  - 同じスレッドへの書き込みはまとめる

GetDの通信イメージ (2/2)

- データの書き込み
  - 同じスレッドへの書き込みはまとめる

4. 局所性を用いた高速化

GetDを用いてCCを再実装

```
for (;;) {
  gr = 0;
  upc_forall (i=0;i<m;i++) {
    u = E[i].u; v = E[i].v;
    GetD({u, v}, Dout)
    if (Dout[u] < Dout[v]) {
      SetD(Dout[v], Dout[u]);
      gr = 1;
    }
  }
  ...
}
```

※ 論文からの推測。 shot-cutting 部分でもGetD/SetDを使用している(詳細略)

4. 局所性を用いた高速化

評価

- 1 (threads/node) * 256 (node)

4. 局所性を用いた高速化

キャッシュ性能の改善

- ノード間の場合と同様の手法を用いる
  - ただし、プロセッサ数 t で分割した後、さらに t' 個のブロックに分割

- 1ノード16スレッドでの結果
- X軸はパラメータ t'
- Y軸は手法適用前と比較した実行時間

いくつかの改善 (1/2)

- compact:
  - 明らかに不要な辺を除く
- offload:
  - アルゴリズム上、D[0]へのアクセスが増大
  - D[0] は常に 0 なのでアクセス削減可能
- circular
  - GetD内の各ループは 0 から n の順に通信する
  - 順序をスレッド id からに変更することにより通信の集中を防ぐ

いくつかの改善 (2/2)

- localcpy:
  - 明らかにスレッド内でコピーできるデータは
    ローカルコピーする
- id:
  - グローバルメモリがどのスレッドに属しているか
    判定するのに時間がかかる (UPC)
  - 最初に計算しておき、後は再利用する
- RDMA:
  - RDMAが利用できる部分はRDMAを用いる

5. いくつかの改善

評価

- 8 (threads/node) * 16 node

5. いくつかの改善

評価

- SMP実装と比較して最大2.2倍の速度向上

まとめ

- SMPクラスタにおいて
  UPCを用いて並列グラフアルゴリズムを実装
  - いくつかの改良を施すことで、
    SMP向けのアルゴリズムをPGAS上で高速に実装
    できることを示した
  - SMPアルゴリズムの通信効率/キャッシュ効率を
    改善するための collective を提案した