IRD
Institut de recherche pour le développement

Optimisation du compilateur de Green-Marl pour l'analyse des réseaux sociaux

Journées Compilation

MESSI NGUELE Thomas

Sous la co-direction de:

Pr. Jean-François MEHAUT
UJF-CEA/LIG

Pr. Maurice TCHUENTE
UY1/LIRIMA-UMMISCO

Perpignan, 21 septembre 2015

Plan

1. Un DSL pour les réseaux sociaux, pourquoi ?
2. Compilateur de Green-Marl
3. Quelques optimisations possibles
4. Conclusion

Définition et caractéristiques des réseaux

Réseau social

- ensemble d'entités (individus) interconnectées par des liens (amitié).
- modélisation à l'aide des **graphes**
  - **très grand nombre de noeuds**, (des milliers, des millions, voir milliard)
  - **faible densité de liens**, un noeud est lié en moyenne à 150 autres noeuds.
- génère une grande quantité de données (nécess. un grand volume de calcul).

Quelques propriétés des graphes sociaux

- structure en communautés,
- distribution de degré fortement hétérogène,
- ...

Exemples d'analyse des réseaux sociaux

Analyse centrée sur :

1. **les nœuds** : déterminer l'importance d'un nœud
   - PageRank, Centralité
2. **les communautés** : sous-ens. de nœuds plus connectés à l'inter. qu'à l'exter.
   - détection des communautés, partitionnement
3. **la structure** : comprendre les lois de l'évolution du réseau
   - prévision des liens, complétion des liens

Analyse parallèle : démarche d'implémentation

Après avoir conçu son algo (séquentiel)

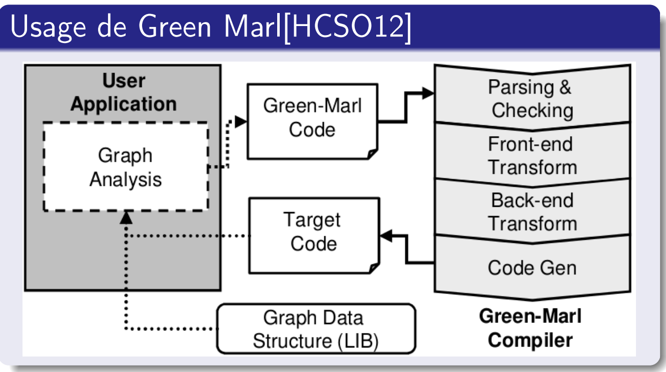
Usage des modèles de programmation parallèle existant

-
  - Étape 1 : Prototypage avec python ou R (pour validation)
  - Étape 2 : Implémentation en C ou C++ (pour les perf.)
  - Étape 3 : Parallélisation puis implémentation avec
    - OpenMP : détection des communautés sur multi-cœur [RBM12]
    - CUDA : détection des communautés sur **GPU** [SN11]
    - Posix thread, MPI, ...
- **Connais. néces. en fouille des réseaux sociaux et en parallélisme.**

Usage des DSLs parallèles de graphe

- Étape 1 : Implémentation avec le formalisme du DSL
- **Approche plus intuitive pour l'algorithmicien, une seule étape**

Usage de Green Marl[HCSO12]



1. Extraction de la partie traitant de l'analyse des graphes
2. Implémentation en Green-Marl
3. Compilation (analyse synt. et séman., optimisation, génération de code)
4. Intégration dans l'application initiale

Élements du langage

```
Procedure pagerank(G: Graph, e,d: Double, max: Int; pg_rank: Node_Prop<Double>)
{
    Double diff;
    Int cnt = 0;
    Double N = G.NumNodes();
    G.pg_rank = 1 / N;
    Do {
        diff = 0.0;
        Foreach (t: G.Nodes) {
            Double val = (1-d) / N + d*
                Sum(w: t.InNbrs) {
                   w.pg_rank / w.OutDegree()} ;

            diff += | val - t.pg_rank |;
            t.pg_rank <= val @ t;
        }
        cnt++;
    } While ((diff > e) && (cnt < max));
}
```

$$PgRank(t) = (1-d) + d\sum_{w->t}(\frac{PgRank(w)}{w.outDeg})$$

- **Itérations**
  - Foreach, for
  - BFS, DFS
  - d'autres semblables au c
- **Réduction**
  - Min, Max, Sum, Product, Count

- **Types de données et collections**
  1. types spécifiques :
     - DGraph, Ugraph ;Node, Edge ; Node/Egde properties
  2. collections :
     - Set(élts uniq., pas ordonnés) ; Order(élts uniq. et ordonnés) ; Sequence(élts pas uniq. mais ordonnés).

Analyse syntaxique et sémantique

- similaire au cas des langages généralistes
- mais plus simplement (domaine restreint)

```
Int y=0;
Foreach(s:G.Nodes)(s.C>3){
  Foreach(t:s.Nbrs){
    Int x = y * s.B;
    s.A += t.B * X @ t;
  }
  s.B = 4;
}
```


Dans le cas des boucles

- Commence par le corps ensuite l'entête (voir schema ci-dessus)
- construction puis confrontation des ens. des variables lues, écrites et réduites
- si conflit read-write alors warning (cas de B dans l'exemple)
- si conflit read-reduce, write-reduce, reduce-reduce alors erreur

Transformation indépendante de l'architecture

Affectation en groupe

Code source

```
Node_Set S(G);//elem uniq
Node_Seq Q(G);//elem non uniq
S.A = S.A + S.size();
Q.A = Q.A + Q.size();
```

Après enlèv. du sucre synt.

```
Foreach(s:S.Items)      // par
  s.A = s.A + S.Size();
For(q:Q.Items)          // seq
  q.A = q.A + Q.size();
```

Transformation indépendante de l'architecture(2)

Réduction

Code source

```
Int y = Sum(s:G.Nodes){
  Product(t:s.Nbrs)(f(t)){
    X(t,s)
  }
};
```

Après enlèv. du sucre synt.

```
Int y;
Int _s0 = 0;
Foreach(s:G.Nodes){
Int _p1 = 1;
  Foreach(t: s.Nbrs)(f(t))
        _p1 *= X(t,s) @ u;
_s0 += _p1 @ s;
}
y = _s0;
```

Transformation indépendante de l'architecture(3)

Fusion de boucle

Code source

```
Foreach(s: G.Nodes)(f(s))
    s.A = X(s.B);
Foreach(t: G.Nodes)(g(t))
    t.B = Y(t.A)
```

Après fusion

```
Foreach(s: G.Nodes){
    if (f(s)) s.A = X(s.B);
    if (g(s)) s.B = Y(s.A);
}
```

Déplacement de déclaration

Code source

```
For(s:G.Nodes) {
    Node_Prop<Int>(G) A;
    ...
}
```

Après déplacement

```
Node_Prop<Int>(G) A;
For(s:G.Nodes) {
  ...
}
```

Transformation indépendante de l'architecture(4)

permutation des arêtes

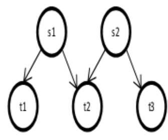
Code source

```
Foreach(t:G.Nodes)(f(t))
  Foreach(s:t.InNbrs)(g(s))
    t.A += s.B;
```

Après permutation

```
Foreach(s:G.Nodes)(g(s))
    Foreach(t:s.OutNbrs)(f(t))
      t.A += s.B;
```

- Nécessite la connaissance du domaine



Fusion de boucle

Code source

```
Node_Set S(G);
Foreach(s: S.Items)
  s.A = x(s.B);
Foreach(t: G.Nodes)(g(t))
  t.B = y(t.A)
```

Après : *Has*() doit être en $O(1)$

```
Foreach(s: G.Nodes){
  if (S.Has(s)) s.A = x(s.B);
  if (g(s)) s.B = y(s.A);
}
```

Sélection de la zone parallèle

```
Foreach(s:G.Nodes){
  InBFS(x:G.Nodes)
     doX(x);
  Foreach(y:s.Nbrs)
    doY(y);
  Foreach(z:G.Nodes)
    doZ(z);
}
```

```
For(s:G.Nodes) {// Seq
  InBFS(x:G.Nodes) // Par
    doX(x);
  For(y:s.Nbrs)// Seq
    doY(y);
  Foreach(z:G.Nodes) // Par
    doZ(z);
}
```

Génération de code

Caractéristiques du code

- Généré en c++ à partir des templ. du backend du compil.
- parallélisme assuré en utilisant openMP (V. infér. 3.1)
  - implicit : transparent au programmeur (G.BC = 0, une affectation).
  - explicit : fait de manière consciente (foreach, BFS).

Structures de données cibles

- Graphe réprésenté avec le format de Yale ou CSR(Compressed Sparse Row)
- *Node_Proprity* et *Edge_Proprity* représenté par des tableaux

Exemple de code généré

Code source

```
Foreach(s:G.Nodes)
  For(t: s.Nbrs)
    s.A = s.A + t.B;
```

Code généré

```
#pragma omp parallel for
for(index_t s = 0; s < G.numNodes(); s++) {
  for(index_t t_=G.edge_idx[s]:t_<G.edge_idx[s+1];t_++){
      // On obtient le noeud à partir des arêtes
      index_t t = G.node_idx[t];
      A[s] = A[s] + B[t];
  }
}
```

Attachement préfentiel : description

Attachment préfrentiel

- $attach_score(x, y) = |\Gamma(x)|.|\Gamma(y)|$
  - $\Gamma(x)$ représente l'ensemble des voisin de x
  - et $|\Gamma(x)|$ est le cardinal de cet ensemble

Algorithme

1: **preferential_attachment**$(Graph\ G < N, E >)$
2: **for all** $x \in N$ **do**
3:     **for all** $y \neq x \in N$ **do**
4:         $pref_attach \leftarrow |G.Neighbor(x)| * |G.Neighbor(y)|$
5:         $print(x, y, pref_attach)$
6:     **end for**
7: **end for**

Attachement préfentiel :implémentation en Green-Marl

```
Procedure preferential_attachment(G:Graph)
{
  Foreach(n1:G.Nodes)
  {
    Foreach(n2:G.Nodes)(n1 < n2)
    {
        Int pref_attach = n1.Degree() * n2.Degree();
        [printf("{(%d,%d),%d}\n",$n1,$n2,$pref_attach)];
    }
  }
}
```

Attachement préfentiel :code Généré

```
 #include "preferential_attachment.h"
void preferential_attachment(gm_graph& G)
{    //Initializations
    gm_rt_initialize();
    G.freeze();
    for (node_t n1 = 0; n1 < G.num_nodes(); n1 ++)
    { #pragma omp parallel for
        for (node_t n2 = 0; n2 < G.num_nodes(); n2 ++)
        { if (n1 < n2)
            {
                int32_t pref_attach = 0 ;
                pref_attach = (G.begin[n1+1] - G.begin[n1])
                               * (G.begin[n2+1] - G.begin[n2]) ;
                printf("{(%d,%d),%d}\n",n1,n2,pref_attach);
            }
}}}
```

Les optimisations envisagées

1. Centrées sur les structures de données en backend (**tâche en cours**)
   - Structure de données graphe adaptée aux graphes sociaux,
   - exploitant leurs propriétés pour réduire le temps d'exécution
2. Centrées sur la parallélisation (**prochaine tâche**)
   - Paralélliser au niveau des tâches et non au niveau des boucles
   - Ordonnancement spécialisé des threads (en fonction du dégré d'un nœud)

Question à 656 fcfa : **quelles autres optimisations ?**

Structure de données exploitant la structure communauté

Objectif

- réduire le nombre de défauts de cache,
- conséquence : réduction du temps d'exécution.

Prémiers résultats[NTM15]

- Gains allant jusqu'à 20% pour les défauts de cache
- et 14% pour le temps d'exécution
- Comparé aux représentations de Yale (CSR), bloc ou liste d'adjacence

Perspective directe

- implémenter dans le compilateur de Green-Marl

Conclusion

- Présentation de Green-Marl et de son compilateur
- Les réseaux sociaux et leurs caractéristiques
- Les optimisations possibles sur le compilateur de Green-Marl

Merci de votre aimable attention!

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun.
Green-marl : a dsl for easy and efficient graph analysis.
In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut.
Exploitation de la structure en communautés pour la réduction des défauts de cache dans la fouille des réseaux sociaux.
Submited in the CRI'2015, Conférence de Recherche en Informatique, decembre 2015.

Jason Riedy, David A. Bader, and Henning Meyerhenke.
Scalable multi-threaded community detection in social networks.
IEEE Computer Society Washington, DC, USA 2012, 18(1) :1619–1628, 2012.
IPDPSW '12 Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum.

Jyothish Soman and Ankur Narang.
Fast community detection algorithm with gpus and multicore architectures.

IEEE International Parallel Distributed Processing Symposium, 2011.