



8. Paradigme Master-Worker

Résumé

L'implémentation du mécanisme de coordination Master-Worker utilise les bibliothèques openMPI et Boost.MPI. L'algorithme est d'abord détaillé puis des expérimentations sont exposées. Enfin la comparaison avec d'autres approches similaires exhibe des résultats intéressants qui dépassent parfois les performances de solveurs perfectionnés tels que cplex ou daopt.

8.1 Description du paradigme Master-Worker

Le paradigme Master-Worker est très connu et largement utilisé dans la communauté du calcul haute performance (HPC). Dans son implémentation la plus simple, un seul processus master coordonne le travail d'un ensemble de processus workers. Les workers n'échangent qu'avec le master et les échanges avec le master se limitent à recevoir le travail et à restituer les résultats, rien entre les deux. Comme on l'a déjà vu, un worker ne *parle* pas en travaillant et le master ne le dérange pas lorsqu'il est occupé. Le rôle principal du master est d'assurer que les processeurs, utilisés par les workers, soient occupés la plus grande partie du temps. De part sa conception, le master assure donc le *load balancing*. Mais il peut être en charge d'autres opérations.

Cependant, comme toutes les communications passent par le master, on peut prévoir un phénomène de goulot d'étranglement du fait d'une augmentation linéaire des échanges avec le nombre de workers. Ceci a pour conséquence de limiter l'échelonnabilité (*scalability*) du programme concrétisée par une *efficacité* décroissant avec le nombre de workers ; par construction le paradigme Master-Worker, implanté dans sa forme la plus simple, présentera un *speedup* par processeur décroissant. Cette limitation peut cependant être dépassée par exemple si le master demande aux workers d'envoyer directement le travail non terminé à ses *collègues*.

En outre, la granularité de niveau nœud du master-worker peut être modifiée dynamiquement pour passer à une granularité de niveau *sous-arbre* où le worker travaillera en toute indépendance tant que le master ne sera pas disponible. La centralisation de l'état de la recherche permet aussi de conserver dans une certaine mesure l'ordre de la recherche du programme séquentiel. On retrouve la nécessité de trouver un compromis entre échanges d'informations qui vont permettre d'éviter aux workers de faire un travail qu'ils n'auraient pas fait en séquentiel et l'overhead de communication décrit au chapitre 6 : Stratégies de parallélisation.

Malgré ses défauts potentiels et la simplicité du son principe, cette approche a été utilisée avec succès pour résoudre des instances difficiles non encore résolues de problèmes MILP¹, en utilisant le solveur CPLEX², de la même manière que les sous-problèmes étaient générés par toulbar2 dans le chapitre précédent, pour générer les sous-problèmes qui sont ensuite traités par une grille de calcul [24].

¹On appelle programme linéaire à variables mixtes, ou Mixed Integer Linear Programming, un programme linéaire contenant à la fois des variables continues donc dans \mathbb{R} et des variables discrètes donc dans \mathbb{Z} . Comme cette catégorie de problèmes généralise les programmes linéaires à variables entières, un MILP est un problème NP-difficile. Toutefois, dans de nombreux cas pratiques, ils sont faciles à résoudre à l'aide de solveurs entiers basés sur des B&B.

²CPLEX est un solveur commercialisé et maintenu par IBM. Son nom fait référence au langage C et à l'algorithme NP-hard du simplexe. Il est composé d'un exécutable et d'une bibliothèque de fonctions interfacées avec divers langages dont python, C, C++ et java : <https://www.ibm.com/analytics/cplex-optimizer>

8.2 Description de l'algorithme

8.2.1 Vue globale

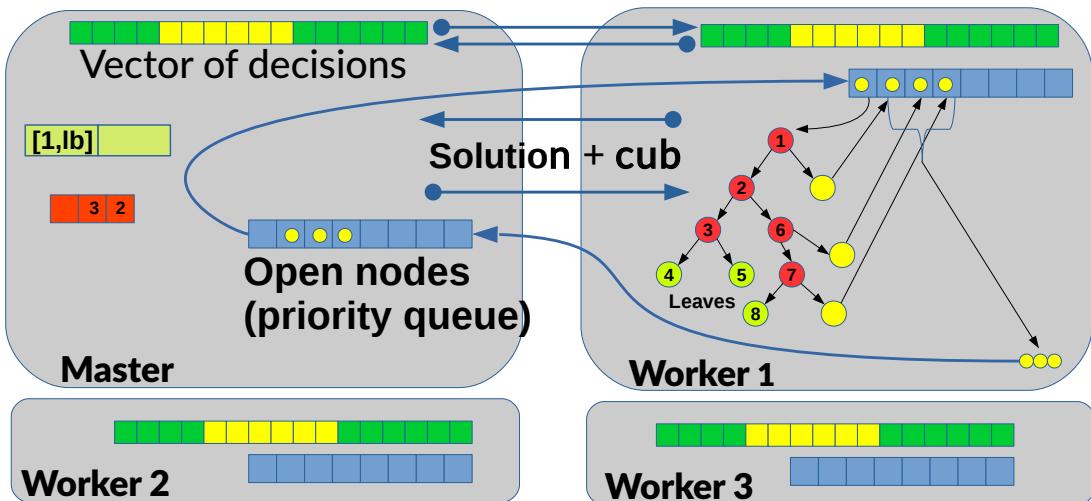


FIGURE 8.1 : HBFS Parallélisé : le master fournit le nœud 1 au worker qui lance un B&B dans le sous-arbre binaire de recherche associé à ce nœud. Les bornes inférieures lb sont propagées dans les nœuds enfants et servent à l'élagage de l'arbre. Le worker transmet au master les nœuds ouverts dès que le nombre maximum de backtracks, ici égal à 3, est atteint.

8.2.2 Algorithm Master

Algorithm 2: Parallel Hybrid Best-First Search : Master

```
/* INITIALISATIONS */  
1 cp := Ø; /* cp = δ vecteur de choice points. */  
2 open := v(cp, lb = clb); /* initialisation du nœud racine */  
3 idleQ := {1,2,...}; /* Au début, tous les workers sont inactifs */  
4 activeWork := Ø; /* map avec les rangs des workers actifs et le lb min */  
   /*  
   /* Tant que clb < cub et qu'il reste du travail à distribuer ou en  
   cours */  
   /* TRAITEMENTS */  
5 while (clb < cub and (open ≠ Ø or activeWork ≠ Ø)) do /*  
   /* Tant qu'il reste du travail et des workers pour le faire */  
6   while (open ≠ Ø and idleQ ≠ Ø) do /*  
     /* le master envoie un nœud, cub et la solution au worker */  
7     isend(node,ub,masterSol); /* isend() non bloquant */  
8     pop(idleQ); /* le worker devient actif */  
9     activeWork[worker] := lb; /* mémorise le lb du worker */  
10    v := pop(open); /* on pop le nœud avec lb minimum */  
    /* Le master attend le message d'un worker quelconque */  
11   recv(mpi :: any_source,tag0,work2); /* recv() bloquant */  
    /* A la réception d'une réponse, le master se remets au travail */  
12   cub := min(cub,cub(worker)); /* mets à jour le cub et la solution */  
13   cp := cpWorker; /* maj avec le vecteur de décisions du worker */  
14   open := openWorker; /* maj avec la file de priorité du worker */  
15   activeWork.erase(worker); /* efface la paire [worker,lb] */  
    /* calculate the min of the lb among the active workers */  
16   minLbWorkers = min(activeWork(worker));  
17   idleQ.push(worker); /* le worker devient inactif */  
18   clb := max(clb,min(lb(open),minLbWorkers)); /* calcule le clb global */  
19   showGap(clb,cub); /* affiche le gap d'optimalité */  
20   print(UB); /* affiche la valeur optimale si elle existe */  
    /* termine tous les processus workers */  
21 return (clb,cub); /* retourne le gap au hbfs parallel */
```

8.3 Implémentation de l'algorithme

8.3.1 Bibliothèques utilisées

L'implémentation du Master-Worker a été effectuée en utilisant le standard MPI via son implémentation openMPI et l'interface C++ Boost associée : https://www.boost.org/doc/libs/1_71_0/doc/html/mpi.html.

8.3.2 Utilisation du hbfs parallèle

Il suffit d'ajouter l'option -para à la ligne de commande de toulbar2 :

```
./toulbar2 -para problem.wcsp
```

8.3.3 Description synoptique

Les deux fichiers principaux ci-dessous sont accessibles sur GitHub :

- <https://github.com/toulbar2/toulbar2/blob/kad2/src/search/tb2solver.cpp>
- <https://github.com/toulbar2/toulbar2/blob/kad2/src/search/tb2solver.hpp>

Dans le fichier.hpp, la class Work définit les messages échangés entre master et workers. Un constructeur permet de formater les messages envoyés par le master aux workers, un autre se charge des communications dans l'autre sens.

Dans le fichier.cpp, la méthode pair<Cost, Cost> Solver : :hybridSolvePara(Cost clb, Cost cub) contient l'implémentation parallèle avec Boost.MPI de l'algorithme HBFS obtenu à partir de l'algorithme séquentiel de la méthode pair<Cost, Cost> Solver : :hybridSolve(Cluster *cluster, Cost clb, Cost cub).

Le master comme les workers envoient les messages en mode non-bloquant via la fonction isend()³ et les reçoivent en mode bloquant via recv().

Le master comme les workers possèdent leur propre file *open*. Le master distribue les nœuds aux workers et la solution courante, y compris la valeur courante *cub*, qui se chargent d'effectuer le B&B avec parcours DFS borné par le nombre de backtracks (DFBB). Les bornes inférieures locales *lb* sont calculées pour chaque nœud. Comme dans tout B&B, les *lb* servent à élaguer l'arbre de recherche. Les workers renvoient le travail non terminé au master sous la forme de nœuds ouverts.

Le master comprend deux boucles while imbriquées. La boucle externe assure la terminaison du programme ce qui n'est pas trivial dans le cas de la programmation parallèle. La boucle interne permet au master de distribuer le travail aux workers.

Pour des raisons d'optimisation spatiale un vecteur de *choice points*, ou vecteur de décisions, noté *cp*, est utilisé. Les nœuds en tant qu'objets possèdent comme attributs, outre le coût local *lb*, deux index qui pointent sur une partie du vecteur *cp*. La séquence de décisions associée au nœud n'est donc pas directement mémorisée dans le nœud, ce qui conduirait à des duplications de données. En effet, dans l'arbre de recherche, certains nœuds sont appelés à partager une partie de leur chemin.

³isend() correspond à un *immediate send* donc non bloquant. La fonction recv(), non préfixée par la lettre i, est bloquante ; le processus attend de recevoir un message pour poursuivre l'exécution du code.

8.3.4 Algorithme Worker

Algorithm 3: Parallel Hybrid Best-First Search : Worker

```
/* boucle infinie */  
1 while (1) do  
2   cpWorker := Ø ; /* vecteur de choice points ou décisions du worker.  
*/  
3   openWorker := Ø ;           /* initialisation de la PQ du master */  
4   recv(master,tag0,work);    /* Le worker attend le travail du master */  
/* A la réception d'un nœud, le worker se mets au travail */  
5   cub := min(cub,cub(master)) ; /* mets à jour le cub et la solution */  
6   cpWorker := cp[first,last] ; /* maj avec les décisions associées au nœud  
*/  
7   openWorker.push(node) ;      /* maj sa file de priorité avec le nœud du  
master */  
/* le DFS B&B mets à jour openWorker avec les nœuds ouverts  
produits */  
8   cub :=DFS(Av,cub,Z);  
9   NodesRecompute := NodesRecompute + v.depth ;  
10  clb := max(clb,lb(open)) ;           /* calcul du clb global courant */  
11  if (NodesRecompute > 0) then  
    /* heuristic Z adaptatif */  
    if (NodesRecompute/Nodes > β and Z ≤ N) then Z := 2 × Z;  
    else if (NodesRecompute/Nodes < α and Z ≥ 2) then Z := Z/2;  
    /* Le worker envoie les nœuds produits, son ub, la nouvelle  
    solution si elle est améliorante et son identifiant. isend() ->  
    envoi non bloquant */  
14  isend(cpWorker,openWorker,newWorkerUb,workerRank,workerSol)
```

8.4 Expérimentations

8.5 Comparaison entre EPS et Master-Worker

Le paradigme Master-Worker est meilleur dans tous les cas comme le montre la table 8.5.

Instance	<i>n</i>	<i>d</i>	S M-W	E M-W (%)	S EPS	E (%) EPS
graph11	340	44	2.22	9.24	0.25	1.05
capmp1	400	200	2.10	8.76	1.63	6.79
scen06	100	44	21.20	88.34	2.16	9.01
pedigree18	1184	5	11.18	51.03	1.32	5.50
nug12	12	12	20.14	83.92	5.63	23.46
404	100	4	16.25	67.70	5.09	21.21

TABLE 8.1 : Speed-up S et Efficacité E pour le Master-Worker(M-W) et l'*Embarrassingly Parallel Search* (ESP). Expérimentations faites sur serveur 24 cœurs. Le nombre de variables *n* et la taille maximale des domaines *d* donnent une idée de la complexité du problème.

Comparaison de toulbar2 avec HBFS séquentiel et HBFS parallèle

Les expérimentations ont été réalisées sur des serveurs 24 cœurs (Intel Xeon E5-2680/87 at 2.50/3GHz and 256 GB) ou sur la platefome GenoToul (64-core nodes of Intel Xeon E5-2683 at 2.10GHz).

Les premiers résultats présentés dans le tableau 8.2 montrent des performances très variables en fonction du problème mais globalement bonnes et toujours meilleures que celles de toulbar2 séquentiel.

Quand ce n'est pas le cas, on peut invoquer une fraction séquentielle du programme non négligeable comme pour le problème capmp1 et invoquer le speed-up théorique de Amdahl pour interpréter les résultats faibles dus à la durée de pré-traitement et de chargement du programme.

Cependant, un autre aspect semble influer sur les performances, celui des temps d'attente du master et des workers. On observe des performances plus faibles en terme de speed-up si les temps d'attente du master comme des workers sont élevés. Ainsi, pour scen06 le temps d'attente des workers est de l'ordre de 0.1s tandis que le master attend les réponses des workers durant 37 secondes. Pour graph11, les workers attendent entre 7 et 25s et le master 122s pour un temps d'exécution de 125s. Le master est donc bien disponible. Les temps de communications ou les temps d'accès et d'écriture dans la mémoire pourraient être invoqués pour expliquer des speed-ups faibles.

Instance	Serial Time (s)	Parallel time (s)	speedup	Efficacité (%)	pre time
graph11	323.12	125.55	2.57	10.72	1.90
capmp1	173.27	82.23	2.11	8.78	6.80
scen06	1026.48	39.22	26.18	109.07	0.16
pedigree18	6.76	0.57	11.86	49.42	0.10
nug12	201.40	8.75	23.02	95.90	0.004
404.wcsp	34.66	1.30	26.66	111.09	0.005

TABLE 8.2 : Exécutions sur serveur 24 cœurs. Le temps CPU est utilisé pour le calcul des speed-ups. La colonne Parallel time indique le temps CPU total. La colonne **pre time** indique la durée du préprocessing.

Eléments nécessaires à la reproduction des résultats

- Le problème graph11 est exécuté avec l'option -A pour utiliser la méthode d'arc cohérence : [Virtual Arc Consistency \(VAC\)](#),
- Options toulbar2 pour le problème pedigree18 : -A -O=-3 -p=-8,
- Exécution en parallèle : mpirun -np 24 ./toulbar2 -para problem.wcsp,
- Exécution séquentielle : ./toulbar2 problem.wcsp,
- Exécution faite sur serveur 24 cœurs de l'unité MIAT : sullo et enfer avec tous les processeurs libres,
- Efficacité = $100 * (\text{speedup}/24)\%$.

Dans les Tables 8.3 and 8.4, on trouvera les temps en secondes nécessaires à la résolution et à la preuve d'optimalité d'instances de type Warehouse et Conception de protéines par ordinateur (CPD). Les tests ont été automatisés sur les serveurs 24 cœurs. Les exécutions sont arrêtées au bout de 1 heure. On notera que la virgule est le séparateur des milliers et non décimal.

On constate dans la table 8.3 une amélioration des performances dans presque tous les cas de figure pour le HBFS parallèle (PHBFS). Les speed-ups restent cependant faibles.

Instance	<i>n</i>	<i>d</i>	Time (sec.)		Speed-up
			HBFS	HBFS-24	
capmo1	200	100	10.92	5.14	2.12
capmo2	200	100	1.80	2.04	0.88
capmo3	200	100	6.09	3.73	1.63
capmo4	200	100	4.36	3.21	1.36
capmo5	200	100	2.69	2.58	1.04
capmp1	400	200	172.57	80.64	2.14
capmp2	400	200	95.15	61.35	1.55
capmp3	400	200	75.04	56.25	1.33
capmp4	400	200	107.86	78.4	1.38
capmp5	400	200	81.15	52.9	1.53
capmq1	600	300	679.43	412.52	1.65
capmq2	600	300	841.80	503.64	1.67
capmq3	600	300	647.67	431.47	1.5
capmq4	600	300	1,093.55	514.42	2.13
capmq5	600	300	1,388.41	701.61	1.98

TABLE 8.3 : Benchmark warehouse [16] où *n* désigne le nombre de variables du problème et *d* la taille du plus grand domaine de ces variables.

Dans la table 8.4, on voit que PHBFS a résolu 3 instances non résolues par HBFS.

Instance	<i>n</i>	<i>d</i>	Time (sec.)		Speed-up
			HBFS	HBFS-24	
1xaw	107	412	721.43	568.50	1.27
3lf9	120	416	407.28	407.92	1
5dbl	130	384	122.84	171.82	0.71
5e10	133	400	147.73	198.23	0.75
5e0z	136	420	148.26	193.41	0.77
5eqz	138	434	3,366.11	1,049.11	3.21
1dvo	152	389	622.03	463.29	1.34
4bxp	170	439	327.46	395.16	0.83
1is1	185	431	-	2,545.82	-
2gee	188	397	797.22	863.64	0.92
5jdd	263	406	-	2,758.98	-
3r8q	271	418	1,605.30	1,294.97	1.24
1f00	282	430	-	2,140.40	-

TABLE 8.4 : Benchmarks *Computational Protein Design* (CPD) [22]. Les tirets indiquent que le problème n'a pu être résolu en moins d'une heure.

8.5.1 Comparaison avec des travaux similaires

Des expérimentations sur d'autres instances difficiles de conception de protéines ont montré des speed-up intéressants.

Dans la table 8.5, on compare les solveurs cplex, daoopt et HBFS sur le benchmark *Linkage*. Les résultats du solveur daoopt sont obtenus sur un cluster de dual core 2.67 GHz Intel Xeon X5650 6-core CPUs et 24 GB de RAM. Ici, la meilleure scalabilité de PHBFS permet d'obtenir de meilleurs résultats sur cluster sur les instances pedigree31 et 44 mais PHBFS est dominé par cplex sur les instances pedigree19 et 51.

Le solveur daoopt est loin derrière mais avec des speed-ups et efficacités élevés.

	pedigree19	pedigree31	pedigree44	pedigree51
<i>n</i>	793	1,183	811	1,152
<i>d</i>	5	5	4	5
cplex	790	59.30	6.35	36.23
//10	191 (4.14)	9.00 (6.59)	2.48 (2.56)	9.43 (3.84)
//30	75 (10.53)	7.17 (8.27)	2.69 (2.36)	5.34 (6.78)
daoopt	375,110	16,238	95,830	101,788
//20	27,281 (13.75)	1,055 (15.39)	6,739 (14.22)	6,406 (15.89)
//100	7,492 (50.07)	201 (80.79)	1,799 (53.27)	1,578 (64.50)
HBFS	3,126	4.34	39.72	1,608
//10	434.27 (7.20)	1.51 (2.87)	6.08 (6.53)	179.22 (8.97)
//20	227.02 (13.77)	1.39 (3.12)	3.18 (12.49)	72.30 (22.24)
//100	119.43 (26.17)	0.97 (4.47)	1.64 (24.22)	31.40 (51.21)

TABLE 8.5 : Benchmark *linkage* [8] avec divers nombre de coeurs. Entre parenthèses, sont indiqués les speed-ups.

On voit dans la table 8.6 que HBFS parvient à résoudre toutes les instances contrairement à CPLEX et que PHBFS, utilisé avec 10 cœurs, améliore encore les résultats.

Instance	n	d	cplex	cplex-10	HBFS	HBFS-10	Speed-up
1UBI	13	198	-	-	1,023	214.02	4.78
2DHC	14	198	-	-	8.2	5.83	1.41
2DRI	37	186	-	-	135.5	30.00	4.52
1CDL	40	186	-	-	392.6	54.95	7.14
1CM1	42	186	-	6,177	6.6	6.11	1.08
1BRS	44	194	-	-	555.3	107.86	5.15
1GVP	52	182	-	-	596.1	185.75	3.21
1RIS	56	182	-	-	129.7	36.23	3.58
3CHY	74	66	-	5,259	88.7	20.71	4.28

TABLE 8.6 : Autre benchmark CPD [2]. Un tiret indique que la méthode n'est pas parvenue à prouver l'optimalité en moins de 9000s.

8.6 Bilan

La version Master-Worker de PHBFS produit des résultats intéressants qui améliorent la plupart du temps les performances du HBFS séquentiel mais surtout dépassent dans certains cas des solveurs perfectionnés tel que `cplex`⁴ ou `daoopt`⁵.

⁴Solveur activement développé par IBM : <https://www.ibm.com/analytics/cplex-optimizer>

⁵<https://github.com/lotten/daoopt>



9. Conclusion

L'objectif de ce stage, qui s'est déroulé du 18 mars au 15 septembre 2019 à l'INRA Toulouse, a consisté à paralléliser HBFS un algorithme séquentiel de recherche arborescente implémenté en C++ dans [toulbar2](#), un solveur de réseaux de fonctions de coûts représentables sous la forme d'hypergraphes.

Après une première étape bibliographique (1.5 mois), de prise en main des outils et de [toulbar2](#) (0.5 mois), la piste EPS a été explorée durant 1 mois comprenant l'analyse du code, sa modification pour produire les sous-problèmes et les expérimentations et tests associés. La piste Master-Worker a pris environ 2 mois comprenant l'analyse du code, le choix de solutions techniques, les tests, les expérimentations et comparaisons avec d'autres approches utilisées dans des solveurs tels que [cplex](#). La rédaction de ce rapport et d'autres documents, notamment pour le séminaires des stagiaires à l'INRA, a nécessité 1 mois.

La démarche utilisée a consisté à explorer des pistes de parallélisation, à en choisir deux : la piste EPS et la piste Master-Worker et à les explorer c'est à dire à réaliser les développements, les tests, les expérimentations, la comparaison et l'analyse des résultats.

HBFS a finalement été parallélisé avec succès en donnant des résultats probants qui feront l'objet d'une présentation à la 25ème conférence CP 2019¹ sur la programmation par contraintes.

Des améliorations pourraient cependant être apportées :

- En terme d'échelonnableté avec l'aide d'autres paradigmes déjà cités dans ce rapport² : *Superviseur-Worker, Multiple-Masters-Worker, Master-Hub-Worker*
- En terme de speedup, en parallélisant les pré-traitements si toutefois c'est possible,
- En parallélisant l'algorithme : Backtracking Tree Decomposition HBFS (BTD-HBFS).

¹<https://cp2019.a4cp.org/>

²Le présent rapport est publié sur <https://github.com/kad15/SandBoxToulbar2/blob/master/kad/rapport.pdf>