Prise en compte de tâches non-prioritaires dans l'ordonnancement batch

Tchimou N'takpé*, Frédéric Suter+
*Université Nangui Abrogoua, Abidjan 02 Côte d'Ivoire
tchimou.ntakpe@gmail.com / ntakpeeul_sfa@una.edu.ci
+Centre de Calcul de l'IN2P3, Villeurbanne, France
Frederic.Suter@cc.in2p3.fr

Abstract

Les ressources de calcul sont généralement accédées au travers d'un gestionnaire de ressources dont l'objectif principal est de terminer au plus tôt les tâches qui lui sont soumises tout en garantissant une utilisation maximale des ressources et une bonne équité entre utilisateurs. Certains utilisateurs peuvent néanmoins ne pas avoir besoin que leurs tâches finissent au plus tôt mais seulement avant une certaine *deadline*.

Dans cet article, nous considérons la prise en compte du caractère non-prioritaire de telles tâches dans la détermination de l'ordonnancement. Nous proposons un algorithme de faible complexité capable de tirer profit de ce degré de liberté supplémentaire. Puis nous évaluons par simulation l'impact de cet algorithme sur l'ordonnancement produit et la qualité de service rendue aux utilisateurs.

1 Introduction

Afin de garantir l'équité entre leurs utilisateurs et maximiser l'utilisation de leurs ressources, la plupart des centres de calculs utilisent un gestionnaire de ressources, ou *batch scheduler*. Si différents outils, libres ou commerciaux existent, tels que OAR [2], SLURM [10] ou Torque [9], tous reposent sur de grands principes communs. Ces gestionnaires de ressources utilisent notamment des algorithmes simples mais efficaces garantissant un bon passage à l'échelle. Ainsi, la plupart des ordonnanceurs *batch* utilisent la politique du "premier arrivé premier servi" ou *First Come First Served* (FCFS), combinée à des techniques de *backfilling* afin de réduire les temps d'inactivité. Cette volonté de conserver des algorithmes de faible complexité implique que la gestion de l'équité et l'optimisation de l'utilisation des ressources est généralement confiée à des mécanismes externes, tels que les quotas ou les priorités. Enfin, l'objectif commun de la plupart des gestionnaires de ressources est de servir, et donc de faire terminer, au plus tôt les tâches qui leur sont soumises.

Dans cet article, nous faisons l'hypothèse que certains utilisateurs peuvent ne pas être intéressés par ce que leurs tâches finissent au plus tôt mais seulement avant une certaine *deadline*. Par exemple, la complétion de tâches soumises le soir peut n'arriver que le matin suivant sans gêner l'utilisateur. Il en va de même pour des périodes plus longues telles que les week-ends ou les congés. Cette hypothèse offre un degré de liberté supplémentaire à l'ordonnanceur qui peut traiter de telles tâches de façon *non-prioritaire* et ainsi optimiser le placement de tâches plus prioritaires.

Après avoir présenté quelques travaux connexes dans la section 2, nous proposons dans la section 3, un algorithme de faible complexité, et par conséquent extensible et utilisable en production, permettant de prendre en compte ce principe de *deadline*. Nous évaluons également l'impact de cette distinction entre tâches prioritaires et non-prioritaires sur la qualité globale de l'ordonnancement produit dans la section 4. Enfin, la section 5 conclura cet article et présentera quelques perspectives.

2 Travaux reliés

La plupart des ordonnanceurs de batch traitent les tâches dans leur ordre d'arrivée dans le système. Afin de combler les temps d'inactivités sur les ressources causés par les différents placements, cet ordonnance-

ment est généralement complété par une technique dite de *backfilling* qui consiste à avancer l'exécution d'une ou plusieurs tâches afin de combler ces "trous".

Dans l'algorithme EASY [7], l'ordonnanceur dispose de prédictions des temps d'exécution des tâches soumises au système, ce qui permet de prédire la date d'exécution de la première tâche de la file d'attente, et uniquement de cette tâche, en fonction de l'occupation des ressources. Lorsque cette tâche ne peut pas démarrer immédiatement son exécution parce qu'il n'y a pas suffisamment de processeurs libres, l'algorithme calcule sa date de début d'exécution. Ensuite, il parcourt la file d'attente et toute tâche qui peut démarrer immédiatement son exécution sans retarder celle en tête de la file, est utilisée pour le *backfilling*.

L'algorithme *Conservative backfiling* [5, 8] est une alternative ayant des performances similaires à EASY, tout en étant moins agressif. Le principe est de réserver immédiatement des processeurs pour toute tâche soumise au système. L'algorithme trouve le premier "trou" (*slot*) dans lequel la nouvelle tâche peut s'exécuter grâce aux prédictions des temps d'exécution.

Dans ces deux algorithmes, lorsque le temps d'exécution d'une tâche est sous-estimé par l'utilisateur de la plate-forme l'ayant soumise, le système interrompt son exécution dès que son échéance prévue est atteinte. Cette situation doit donc être évitée, ce qui conduit généralement à une surestimation du temps d'exécution par les utilisateurs. En cas de surestimation, quand une tâche termine son exécution plus tôt que prévu, l'algorithme cherche à avancer autant que possible les tâches de la file d'attente.

Dans [6], les auteurs utilisent une métaheuristique basée sur la recherche tabou pour améliorer les performances des ordonnancements. Toute tâche arrivant dans le système est initialement insérée dans la file d'attente selon le principe de *Conservative backfiling*. Ensuite, toutes les cinq minutes, l'algorithme essaye d'optimiser l'ordonnancement des tâches de la file d'attente en minimisant une fonction objectif dépendant de l'équité entre les utilisateurs et des *slowdowns*, des temps de réponse et des temps d'attente des tâches. Il faut noter que cette réorganisation régulière des ordonnancements ne permet pas de prédire les dates de fin d'exécution des tâches, ce qui peut s'avérer gênant pour certains utilisateurs.

3 Algorithme

Dans cette étude, nous considérons que la plate-forme cible est utilisée en espace partagé pour exécuter des tâches non préemptives et que le partage des ressources réseaux n'influe pas sur le temps d'exécution des tâches. Nous supposons également que la plate-forme est totalement homogène, aussi bien du point de vue des processeurs que de son réseau d'interconnexion. Ainsi, une tâche s'exécutant sur 4 processeurs pourra indifféremment être allouée sur un ensemble contigu (p_1, p_2, p_3, p_4) ou disjoint (p_1, p_6, p_8, p_{22}) de processeurs. Cette simplification nous permet de ne pas avoir à gérer le problème de contiguïté des allocations pour le placement des tâches. De plus, les traces utilisées en section 4 ne fournissent en effet aucune information sur la sensibilité des tâches à la contiguïté des allocations. La présence de tâches nécessitant des allocations contiguës ajouterait une contrainte supplémentaire sur leurs placements, mais ne remet pas en cause notre étude.

Le placement d'une tâche T_i consiste à lui attribuer un ensemble de processeurs sur lesquels elle s'exécutera et décider de sa date de début d'exécution st_i (*start time*). Nous distinguons deux types de tâches. Une tâche prioritaire est une tâche qui, une fois soumise, se voit attribuer un placement définitif qui vise à minimiser sa date de fin d'exécution ct_i (*completion time*). Une tâche non prioritaire dispose d'une *deadline* d_i telle que son temps de séjour sur la plate-forme peut être largement supérieur à son temps d'exécution wl_i (*walltime*). Le temps de séjour d'une tâche est la différence entre sa date de fin d'exécution ct_i et sa date de soumission rl_i (*release time*). Les tâches non prioritaires peuvent être retardées, c'est-à-dire que leurs placements peuvent être recalculés tant que leurs dates de fin d'exécution prévues respectent leurs *deadlines* respectives.

Notre algorithme utilise deux listes pour maintenir les tâches en attente d'être exécutées. La première liste contient toutes les tâches dont les placements sont définitifs. Par la suite nous appellerons cette liste, la liste prioritaire. Dans la seconde liste, appelée liste non prioritaire, nous avons uniquement des tâches avec *deadline* dont les placements sont provisoires. En pratique, les ordonnanceurs *batch* fonctionnent en termes de *scheduling round*. Dans un *round* donné, on peut avoir à gérer le placement de plusieurs tâches nouvellement soumises. Mais, puisque dans un *scheduling round* les tâches sont prises en compte dans l'ordre de leurs arrivées, nous pouvons considérer qu'elles arrivent les unes après les autres dans le système. Nous proposons donc un algorithme d'ordonnancement en ligne qui réagit à chaque fois qu'une

tâche arrive.

Losqu'une tâche avec *deadline* est soumise, elle est immédiatement insérée dans la liste non prioritaire et on détermine son placement selon le principe du *Conservative backfilling*. Pour ce faire, nous tenons compte des placements (définitifs ou non) de toutes les tâches en attente d'être exécutées. Cela suppose que la *deadline* est suffisamment grande et qu'elle ne risque pas d'être violée dès la soumission de la tâche.

Lorsqu'une tâche prioritaire arrive dans le système, elle est d'abord insérée dans la liste prioritaire. L'algorithme 1 est alors utilisé pour déterminer son placement définitif. Dans un premier temps, cette procédure annule les placements des tâches de la liste non prioritaire. Ensuite on détermine les tâches de cette liste dont la *deadline* risquerait d'être violée si l'on plaçait la nouvelle tâche prioritaire avant celles-ci. S'il existe de telles tâches, ces dernières sont triées dans l'ordre croissant de leurs arrivées et sont placées successivement et déplacées dans la liste prioritaire. A partir de cet instant, le placement de ces tâches avec *deadline* devient définitif. Enfin, on effectue la réservation de la nouvelle tâche prioritaire.

Algorithm 1 Calculer le placement d'une tâche prioritaire T_p

1: Annuler le placement de toutes les tâches T_i de la liste non prioritaire
2: Construire la liste L_TMP des tâches T_i de la liste non prioritaire dont les *deadlines* d_i risquent d'être violées, puis retirer ces tâches de la liste non prioritaire
3: Recalculer le placement des tâches de L_TMP dans l'ordre croissant de leurs dates de soumission rl_l
4: **Si** (Il existe des tâches T_l de L_TMP dont les *deadlines* sont violées) **alors**
5: Déplacer toutes les tâches T_i telles que $rl_i < rl_l$ de la liste non prioritaire vers L_TMP
6: Recalculer le placement des tâches de L_TMP dans l'ordre croissant de leurs dates de soumission rl_l
7: **Fin Si**
8: Calculer le placement de la nouvelle tâche T_p
9: Déplacer les tâches T_l de L_TMP vers la liste prioritaire
10: Recalculer le placement des tâches T_i de la liste non prioritaire dans l'ordre croissant de leurs dates de soumission rl_i

Avant d'insérer une tâche avec *deadline* dans la liste prioritaire, il peut arriver que le placement déterminé ne permette plus de respecter sa *deadline*, alors qu'elle était respectée à l'issue de son dernier placement provisoire. C'est une situation rare en pratique qui s'explique par le fait que ce placement provisoire qui tenait compte de toutes les autres tâches non prioritaires arrivées plus tôt, mais restées dans la liste non prioritaire, était meilleur qu'un placement ne tenant pas compte de ces tâches. Afin de limiter la complexité de l'algorithme, si une tâche devant migrer vers la liste prioritaire se trouve dans cette situation, on décide de faire migrer également toutes les autres tâches de la liste non prioritaire soumises auparavant. Cela permet de conserver le placement respectant la *deadline*. Pour ce faire, nous utilisons dans notre algorithme une liste temporaire (L_TMP) de tâches retirées de la liste non prioritaire et que l'on s'apprête à migrer vers la liste prioritaire. Le fait de maintenir cette liste, contenant des tâches T_l, triée dans l'ordre croissant de leurs dates de soumissions respectives rl_l est également le choix le plus évident et le plus simple pour garantir le respect des *deadlines*.

La figure 1 illustre notre algorithme. Lorsque la tâche prioritaire T_2 arrive, elle retarde T_1 sans violer sa *deadline*. Mais lorsque la tâche T_4 arrive, la *deadline* de T_1 risque d'être violée. Cette dernière est définitivement placée et passe de la liste non prioritaire L_0 à la liste prioritaire L_1.

Il faut souligner que nous parlons ici de placements définitifs car nous supposons que les prédictions des temps d'exécution des tâches fournies par les différents utilisateurs sont relativement précises. Dans le cas de surestimation importante de ces temps d'exécution, l'algorithme veillera à avancer si possible les tâches en attente.

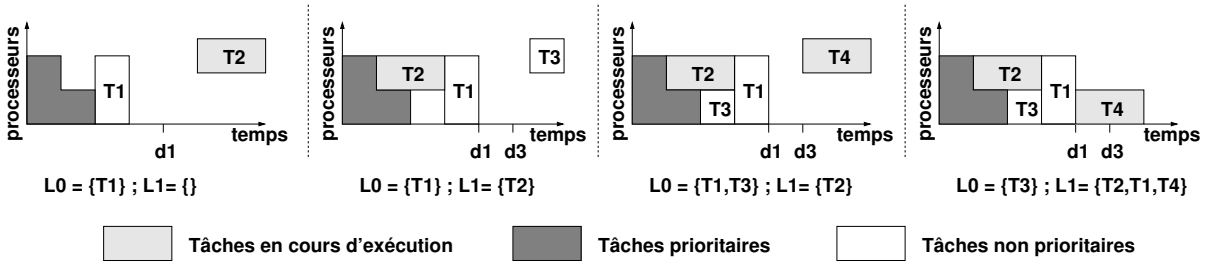

Figure 1: Illustration de l'algorithme DBF.

4 Conditions expérimentales et résultats

4.1 Conditions expérimentales

Pour nos expérimentations nous avons utilisé des scénarios de soumissions de tâches (*workloads*) sur des plates-formes de production et issus de la *Parallel Workloads Archive* [4]. Nous considérons trois *workloads* dont les distributions respectives du nombre de processeurs par tâche et des temps d'exécution par tâche varient dans un large spectre. Leur ordonnancement représente donc un véritable défi.

Ce sont SDSC-BLUE (250 000 tâches soumises d'avril 2000 à janvier 2003 sur une plate-forme contenant 1 152 processeurs), HPC2N (202 871 tâches soumises de juillet 2002 à janvier 2006 sur une plate-forme de 240 processeurs) et SDSC-DS (96 089 tâches soumises de mars 2004 à avril 2005 sur une plate-forme de 1 664 processeurs).

Cette étude n'est qu'une première preuve de concept de l'impact d'une gestion particulière des tâches non prioritaires sur l'utilisation classique du *Conservative backfilling*. Nous avons donc décidé arbitrairement de considérer qu'une tâche sur trois est non prioritaire et dispose d'une *deadline*. A terme, la proportion de tâches non prioritaire sera considérée comme un paramètre à part entière de notre étude.

Dans un premier temps, cette *deadline* est prise telle que le temps de séjour maximum de la tâche sur la plate-forme est le maximum entre 24 heures et 2 fois son temps d'exécution (cas **a**). Ce choix se justifie par le fait que, en analysant les *workloads*, nous observons que le temps moyen d'attente des tâches avec *Conservative backfiling* est de l'ordre de quelques heures. Cette *deadline* permet donc à la plupart des tâches concernées d'avoir largement le temps de s'exécuter. Le choix des 24 heures est également intéressant du point de vue des utilisateurs. Ces derniers se rendent à leur lieu de travail à des heures qui se répètent toutes les 24 heures. Il leur est donc facile de s'adapter au fait qu'ils récupéreront leurs résultats de calcul pour analyse en début de journée, environ 24 heures après la soumission sur une plate-forme parallèle. Nous avons également étudié l'effet de l'augmentation des *deadlines* en les prenant telles que le temps de séjour maximum d'une tâche concernée est le maximum entre 72 heures et 2 fois son temps d'exécution (cas **b**).

Pour nos simulations, lorsqu'à un instant donné, la plate-forme est trop sollicitée de sorte qu'une tâche fraîchement soumise ne peut pas respecter sa *deadline*, alors au lieu de refuser son ordonnancement, on la considère plutôt comme une tâche prioritaire.

Nous avons réalisé des simulations en utilisant *Simbatch* [1] un outil développé au dessus de *SimGrid* [3]. *Simgrid* est un simulateur à événements discrets qui permet de modéliser des plates-formes distribuées, leurs réseaux et des processus qui y interagissent. Ce qui permet toutes sortes de simulations sur ces plates-formes. *Simbatch* permet de simuler aisément un algorithme d'ordonnancement *batch* en l'y intégrant sous forme de *plugin* dans un unique fichier C. Il génère autant de files d'attentes que spécifié dans le fichier de configuration pris en paramètre. Avec *Simbatch* on peut également préciser la file d'attente dans laquelle chaque tâche sera initialement insérée dès sa soumission dans le fichier du *workload* utilisé.

4.2 Résultats des évaluations

Pour évaluer l'impact de notre algorithme, nous utilisons les critères les plus couramment utilisés que sont les temps d'attente wt_i (*wait times*) des tâches T_i ($wt_i = st_i - rl_i$) et leurs *slowdowns* sd_i ($sd_i = (wt_i + wl_i)/wl_i$) que l'on peut qualifier de retards relatifs d'exécution des tâches. Un ordonnancement de

Table 1: Résultats sur les tâches prioritaires

Workloads	SDSC-BLUE			SDSC-DS			HPC2N		
Algorithme	CBF	DBF (a)	DBF (b)	CBF	DBF (a)	DBF (b)	CBF	DBF (a)	DBF (b)
avg_wt	2,36	-21,61%	-46,18%	1,40	-32,86%	-52,14%	3,28	-20,73%	-32,01%
avg_sd	32,13	-18.08%	-45.56%	18,05	-32,08%	-42,55%	127,65	-23,96%	-30,98%

Table 2: Résultats sur toutes les tâches

Workloads	SDSC-BLUE			SDSC-DS			HPC2N		
Algorithme	CBF	DBF (a)	DBF (b)	CBF	DBF (a)	DBF (b)	CBF	DBF (a)	DBF (b)
avg_wt	2,35	-2,98%	-12,77%	1,40	-6,43%	-16,43%	3,28	-6,71%	-9,76%
avg_sd	32,03	-3,09%	-29,41%	17,88	-19,72%	-27,57%	127,14	-17,06%	-21,15%

qualité se traduit par des *slowdowns* moyens (avg_sd) et des temps d'attente moyens (avg_wt) faibles. La réduction des temps d'attente par un algorithme par rapport à un autre permet d'entrevoir que cet algorithme permettra une meilleure utilisation de la plate-forme.

Dans les Tables 1 et 2 nous comparons les résultats de notre algorithme (DBF) au *Conservative Backfilling* (CBF), respectivement en ignorant les tâches non prioritaires et en tenant compte de toutes les tâches. CBF est un algorithme représentatif de ceux couramment utilisés en production et en particulier sur les traces utilisées. Les avg_wt ont été mesurées en heures. En ne considérant que les tâches prioritaires, on observe d'importantes améliorations des performances avec DBF. Cela signifie qu'il y a une nette amélioration de la qualité de service pour les utilisateurs ayant soumis des tâches sans contrainte de *deadline*. En observant les performances moyennes sur toutes les tâches, on note également des améliorations significatives avec DBF. Dans tous les cas, les performances de DBF sont encore meilleures quand on augmente les *deadlines* des tâches non prioritaires (cas **b**). En effet, une tâche dont la deadline est éloignée devrait être plus souvent candidate au backfilling, puisque plus de décisions de placement seront prises avant que sa deadline n'expire.

En plus de l'amélioration des ordonnancements et du strict respect des *deadlines*, notons que les simulations concernant DBF sur un simple ordinateur portable ont pris en moyenne moins de 0,04 secondes pour ordonnancer chaque tâche, pour le *workload* le plus complexe. Notre algorithme est donc un candidat crédible pour les plates-formes de production.

5 Conclusion

Dans cet article, nous avons étudié comment le fait que certains utilisateurs de plates-formes de calcul accédées au travers d'un gestionnaire de ressources puissent spécifier le caractère non-prioritaire de leurs tâches soit pris en compte lors de l'ordonnancement. Nous avons pour cela proposé un algorithme de faible complexité qui améliore l'algorithme classique de *Conservative backfilling* tout en restant déterministe et en permettant de prédire les dates de fin d'exécution (au plus tard) des différentes tâches soumises. Nous avons montré que cet algorithme permettait de réduire le temps de complétion et le *slowdown* moyens de l'ensemble des tâches.

Nous estimons donc que les administrateurs de plates-formes de calcul ont intérêt à mettre en place des politiques incitant de nombreux utilisateurs à soumettre des tels jobs non-prioritaires. Il est cependant encore possible d'améliorer l'ordonnancement de tâches avec *deadlines*, par exemple en adaptant des algorithmes hors ligne dont l'objectif est de minimiser le *makespan*.

Remerciements

Ces travaux sont en partie supportés par l'ANR MOEBUS (ANR-13-INFR-0001).

References

[1] Yves Caniou and Jean-Sébastien Gay. Simbatch: An API for Simulating and Predicting the Performance of Parallel Resources Managed by Batch Systems. In *Euro-Par 2008 Workshops - SGS 2008,*

Revised Selected Papers, pages 223–234, Las Palmas de Gran Canaria, Spain, August 2008.

[2] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A Batch Scheduler with High Level Components. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 776–783, Cardiff, UK, May 2005.

[3] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899 – 2917, 2014.

[4] Dror Feitelson, DAn Tsafrir, and David Krakov. Experience with using the Parallel Workloads Archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[5] Dror G. Feitelson and Ahuva Mu'alem Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, IPPS '98, pages 542–546, 1998.

[6] Dalibor Klusácek and Hana Rudová. Performance and Fairness for Users in Parallel Job Scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 235–252, 2013.

[7] David A. Lifka. The ANL/IBM SP Scheduling System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes on Computer Science, pages 295–303. Springer-Verlag, 1995.

[8] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transaction on Parallel Distributed Systems*, 12(6):529–543, June 2001.

[9] Garrick Staples. TORQUE - TORQUE Resource Manager. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, page 8, Tampa, FL, November 2006.

[10] Andy Yoo, Morris Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, Seattle, WA, June 2003. Springer.