Specification X-Kaapi

V. Danjean, T. Gautier, C. Laferrière

October 20, 2009

Chapter 1

Introduction

1.1 Histoire

Un peu d'histoire pour expliquer où nous en sommes. En 1999, le 22 septembre, avec la thèse de François Galilée, la spécification d'Athapascan était posée: Fork + Shared + les modes et droits d'accès r, w, rw, cw ainsi que les modes postponed (rp,wp, rpwp, cwp). Une implémentation C++ complète a existé (sisi). Le coût a l'exécution était énorme : environ 10000 appels de fonction C pour 1 fork sans paramètre. Ceux-ci s'expliquaient de deux manières: 1/ un algorithme de terminaison distribuée était utilisé pour calcul la fin d'accès à une version pour déclencher les calculs sur la version suivante ; 2/ les choix des structures de données ainsi que leur allocation dans le tas ; 3/ le mauvais couplage entre la couche de communication thread safe Athapscan-0 et l'implémentation d'Athapascan-1 au dessus. Bien qu'en théorie, les algorithmes de cette implémentation distribuée devaient permettre d'avoir du bon speedup, en pratique il fallait avoir des calculs à très gros grain. L'ordonnancement était basé sur des variations autour du vol de travail avec des heuristiques pour permettre de contrôler l'utilisation mémoire, ou bien par ordonnancement de type statique d'un graphe de flot de données mais aucune perf n'a pu être observée.

Quelques semaines plus tard, le 14 décembre 1999, Mathias Doreille montrait dans sa thèse qu'il était possible d'avoir de très bon speedup sur des programmes ordonnancés par une variation d'un algorithme de type ETF et en utilisant une implémentation adhoc très légère directement au dessus de MPI mais incomplète vis-à-vis des spécifications d'Athapascan-1. L'ordonnancement local des messages et du calcul ne permettait pas d'avoir des performances raisonnables sur des programmes du type itératifs (jacobi, poisson) normaux (sans augmentation complexe / artificielle du grain).

En 2004, Rémi Revire a montré la faisabilité d'avoir une implémentation plus efficace permettant à la fois un ordonnancement par vol de travail et par ordonnancement de graphe en se basant sur une implémentation qui, grâce à l'ordre d'exécution quasi séquentiel d'un programme Athapascan, permet de gérer le cycle de vie des tâches et objets partagés en rapport à leurs portées de déclaration : la gestion par pile de l'exécution des programmes séquentiels était retrouvée ! Néanmoins, l'algorithme de terminaison du calcul distribué (non plus de la fin d'utilisation d'une version) était encore mal conçu, mais heureusement n'intervenait que pour déterminer la terminaison globale de l'exécution des processus. Du point de vu des performances, l'implémentation se basait sur l'utilisation de lock pour verrouiller l'accès à certaines données lors des opérations de vols. Le partitionnement statique du graphe utilisait Scotch et était à l'état de prototype instable. L'ensemble était basé sur une couche de communication appelée Inuktitut qui permettait de communiquer par envoi/réception de messages actifs. Laurent Pigeon avait implanté un ensemble d'algorithmes pour la diffusion parallèle de message durant son master. Lors d'un stage d'été, Xavier Besseron avait implanté le côté

communication sur architecture hétérogène en utilisant soit un protocol de type Xdr (eXternal data representation), soit de type ASN. Lors de son stage, Everton Hermann a effectué le portage d'Inuktitut sur Myrinet. Le coût de cette implantation était d'environ 1000 appels de fonction C pour 1 fork (toujours sans paramètre).

En 2005, Kaapi est née des cendres d'Athapascan & Inuktitut : une évaluation synthétique aurait été "ça marche pas trop mal, code de niveau prototype de recherche, mais peut mieux faire" Plus précisément :

- peu de support du côté d'Inuktitut qui avait été conçu pour servir à la conception de Taktuk (1ère version hélas très bogguée en C++) ou des outils ka-XXX pour la diffusion de fichiers.

- 3 protocols de communication dans Inuktitut: active message, write & signal (idem active message mais avec une allocation par la couche de comm des données reçues dans l'espace mémoire utilisateur) et une variante allocate & write & signal bien adaptée à un portage de la couche de comm directement au dessus de CORBA.

- la portée des structures de données dans Athapscan n'était pas encore bien comprise, leur gestion nécessitait encore l'utilisation de locks.

- l'algorithme de vol était décomposé en trois endroits : soit le vol effectif d'une tâche lors d'une requête local, soit lors vol suite à une requête à distance, soit le réveil d'un thread ce qui posait pas mal de problème de "stabilité" de celui-ci en cas de modification (...) En pratique toute tentative de modification d'une partie du vol de tâches ou du choix du reveil d'un thread était source de longues heures de débug...

Beaucoup de concepts avait été validés (couplage ordo statique / vol de travail) mais l'ensemble des sources étaient difficilement abordables par les étudiants arrivant sur le projet. En 2005, il fut donc fait les choix suivants :

1. récupération d'Inuktitut et suppression du code non essentiel: protocol message actif

2. lors du stage d'Everton Hermann il a été vu qu'un fonctionnement de la couche de message par vol de travail permettait simplement d'agréger des messages vers le même destinataire, de plus elle était original dans le sens ou la fenêtre d'aggrégation dépendait de l'activité réseau...

3. une même opération pour le vol de tâche ou le réveil de thread: la fonction de vol prend en entrée un thread et retourne un thread ! Cela a l'avantage d'être simple et uniforme : dans le premier cas, un objet thread est volé et le thread résultat est rempli par la tâche volée, dans le second cas un objet processeur est volé et il retourne le thread à réveiller.

4. la notion de graphe de flot de données (application) doit être découpé du vol de travail (ordonnancement), l'ordonnancement de Kaapi se base donc sur un ensemble de threads qui peuvent être volés, les threads pouvaient être directement du code applicatif ou bien structurés par pile de tâches. En pratique, seul la deuxième possibilité a été utilisé, même à travers les algorithmes adaptatifs de Daouda. Cette structuration a permis de réduire l'utilisation de verrou dans l'implémentation.

5. une conception uniforme des objets pouvant être sérialisés et qui sont représentés par leur "format" : les data ou les threads possèdent un format. La couche API Athapascan génère automatique avec un ensemble d'expression template les objets formats en fonction du type des objets à sérialiser.

6. la capacité d'interagir avec l'exécution par l'envoi de signaux : SIGSTOP, SIGSTOP, ou encore SIGKILL.

7. et enfin, un prototype pour la collection de statistique d'exécution des processus et l'enregistrement d'événement dans des fichiers de trace.

Les derniers points 5/ et 6/ ont facilité en partie l'implémentation des protocoles de tolérance aux pannes : arrêt des activités et sérialisation de facto des threads.

L'interface de programmation de la couche de communication n'a pas bougé depuis 2005, en revanche un développement a été nécessaire pour permettre de faire communiquer un ensemble de processus dynamiques et rendre robuste la panne d'un ou plusieurs processus sans provoquer l'arrêt les autres de manière non contrôlée. De plus, cette couche est "multi-réseau" avec une sélection du réseau à la source en fonction du destinataire qui a l'avantage d'être simple, mais n'est pas forcément très performante.

Cette version de Kaapi a permis de reporter successivement les 3 plugtest 2006 (non officiel dans le sens que nous n'avons pas été classé comme les autres car non java-compliant, mais les résultats étaient bien devant les autres), 2007 et 2008. Pour 2009 : le ETSI Grid@Work plugtest s'oriente vers les aspects uniquement déploiement... on ne participera pas.

Mes critiques sont :

1. code gros devenant gros ne facilitant pas la gestion de l' "histoire" pour les nouveaux arrivants.

2. les perfs de l'aspect purement multi-core / SMP / many-core peuvent être meilleur

3. les choix des routes / des interfaces dans la couche communication doit être amélioré. N'ayant pas de travaux de recherche (thèse) sur ce point, nous n'arrivons pas à nous reposer sur une couche externe "portable" et robuste (grid, cluster, ...) ayant des capacités de dynamicité / robustesse en cas de panne qui soient suffisantes.

4. l'absence d'une interface permettant de gérer la transparence référentielle des objets partagés, i.e. être capable d'accéder en lecture ou écriture aux objets quel que soit le site d'exécution. Ce point est délicat et doit être guidé par certaines décisions d'ordonnancement.

5. l'absence de la notion de collection d'objets distribués (utilisant un support comme décrit dans le point 4/ ci-dessus) permettant la description de la plupart des algorithmes en calcul scientique pour lesquels il serait intéressant d'avoir à manipuler des tableaux multi-dimmensionels d'objets "shared".

6. absence d'une couche "serveurs de stockage fiable" pour les protocoles de tolérance aux pannes... De même que pour 3, il est difficile de trouver du source dans ce domaine qui permettent de bonne performance. Néanmoins une évaluation des outils récents seraient à refaire (la dernière date de 2006-2007).

7. mauvais couplage des algorithmes adaptatifs.

Début 2009: naissance de X-Kaapi pour répondre aux points 2/ et 6/ tout en facilitant le portage de X-Kaapi sur des systèmes embarqués dans le cadre de nos collaborations avec ST, i.e. en utilisant du C !

1.2 Objectifs et problématiques

Les objectifs de X-Kaapi sont de capitaliser le savoir faire que nous avons eu en vu de cibler de nouvelles architectures (pour nous) et de nouvelles applications. Le contexte technologique est induit par des architectures à mémoire hiérarchique composées par des processeurs de type multi-core ou many-core ou du domaine de l'embarqué (MpSoC). Ces processeurs communiquent par de la mémoire ou un réseau. La latence d'accès à la mémoire varie fortement en fonction de la distance entre la données et le cœur qui y accède. L'environnement d'exécution disponible n'est peut-être pas aussi complet que celui d'un PC actuel. En particulier en ce qui concerne l'ordonnancement de threads, la gestion mémoire (pas de mémoire virtuelle, taille limité). Le nombre total de cœur peut être très important.

Les deux architectures cibles initiales sont :

- une bluegene : un nœud est composé d'un ensemble de cœurs partageant une mémoire local, partageant ou non une hiérarchie de caches. Le réseau d'interconnexion est de type grille 3D torique (sous contrainte de routage au bord en fonction d'une réservation partielle). Ce réseau n'est utilisé que pour les communication bi-point. La machine dispose aussi d'un réseau de diffusion (broadcast) et d'un réseau de synchronisation. La taille mémoire par nœud est limité (256MBytes ou 512MBytes). Les possibilités de l'OS sont à découvrir. La couche de communication DCMF disponible est de bas niveau et offre une communication de type messages actifs et lecture/écriture à distance. Les questions qui se posent sont liés à la capacité à traiter des handlers de communication en concurrence avec du calcul, la capacité du système à faire progresser les communications et à ordonnancement plusieurs threads.

- MpSoC : la board ST composé de 4 cœurs ST 231 qui communiquent par des envois de message ou du partage d'objet (dans un sens très particulier) vers, et uniquement vers, le cœur principal ST40. Très peu de mémoire (quelques MBytes), un OS nouveau pour nous (OS21) mais qui possède la notion de task thread et qui permet leur ordonnancement (préemptif ?) sur un cœur ST231. A noter que l'OS du ST40 est un Linux.

En pratique et pour commencer le développement, une architecture du type Idkoiff sera utilisé.

Du point de vu des applications numériques visées, nous considérerons les types de code suivants :

recherche arborescente ces applications sont caractérisés comme très consommatrice de CPU et peu gourmande en ressources réseaux. Elles possèdent un ration calcul / accès mémoire très important.

- QAP
- NQueens

calcul scientifique ces applications sont encore très gourmandes en CPU et possède un ration calcul / accès mémoire assez variable mais faible.

- BLAS
- NAS

calcul sur des flots ici le ratio calcul / accès mémoire est proche de 1.

- STL

simulation interactive

- SOFA

Les problématiques associées sont décrites dans les sections suivantes.

1.2.1 API pour le calcul

Dans Kaapi/Athapascan le modèle imposé est de type graphe de flot de données sur lequel nous avons construit beaucoup d'outils (partitionnement de graphe, protocol de tolérance aux pannes, ...). En revanche, ce modèle est très contraignant pour la programmation et l'absence d'abstraction pour décrire et manipuler des objets partagés contenant d'autres objets partagés (ex: un array of shared lui même shared) est contraignant. De plus, dans le cas des algorithmes à grain fin, la construction d'une représentation abstraite est un facteur pénalisant.

Les questions qui se posent sont :

Q 1. doit-on conserver cette interface de programmation fork/shared ? Comme élément de réponse, notons qu'il est possible d'exécuter des programmes série-parallèle récursif avec un surcoût de l'ordre de 1 (à 2 ?) par rapport au même code séquentiel : cela nécessite éventuellement l'écriture d'un compilateur C -¿ C pour faciliter la génération de code. Cf présentation à LogProg que j'ai faite avant de partir. En pratique cette interface est aussi utilisée pour la construction d'un graphe de tâche en exécutant que les instructions fork d'une tâche, sans exécuter le corps des tâches créées :

Q 2. comment transformer la manière d'interpréter les programmes pour permettre de construire le futur d'une exécution afin d'améliorer l'ordonnancement par un algorithme "statique" ?

Enfin, remarquons que dans le cas des algorithmes adaptatifs, la construction de tâches indépendamment des ressources disponibles est source de surcoût, voir par exemple l'exemple du prefix de Jean-Louis/Daouda. La solution proposée et retenu dans le prototype X-Kaapi est de donner la capacité au programmeur d'expliciter les points où seront fait les extractions de parallélisme. A la charge du système, i.e. de l'algorithme de vol, de faire les bons choix... Une autre solution serait peut-être d'utiliser une approche à la TBB (Daouda pourra me corriger) : dans TBB dans le cas d'une exécution séquentiel, la création de 2 tâches à un niveau de récursion ne se traduit pas forcément par l'exécution des deux tâches. Dans le cas où la première tâche se termine et s'il n'y a pas eu de requête de vol, la seconde est tout simplement annulée afin que le calcul séquentiel puisse se poursuivre.

Pour terminer, et au vu des très bons résultats expérimentaux sur les algorithmes de la STL parallélisés avec X-Kaapi par Daouda :

Q 3. faut-il utiliser seulement l'interface X-Kaapi pour la programmation parallèle ?

En conclusion il serait important que TOUS nous ayons une discussion sur une éventuelle interface de programmation (...).

1.2.2 Runtime pour l'ordonnancement local

Dans Kaapi, le runtime abstrait les ressources de calcul en associant un K-processeur par ressource de calcul. Celui-çi étant en charge de gérer l'ordonnancement des calculs sur cette

ressource. Le calcul est ensuite représenté par un K-thread qui peut s'exécuter sur l'un des K-processeurs en respectant éventuellement des contraintes de placement. Un K-thread peut être volé afin de lui extraire du travail. De même un K-processeur peut être volé pour lui extraire un K-thread. Le voleur exécute le code du traitement de vol en concurrence à l'exécution de la victime.

Dans le prototype X-Kaapi, un K-processeur est lancé sur chacun des cœurs choisis lors du lancement de l'application. Le K-thread déroule les instructions du calcul dans lequel des points de vol ont été insérés. Si un ou plusieurs K-processeurs ont posté une requête de vol vers un même autre K-processeur, le K-thread actif sur le K-processeur victime répond à la requête. La limitation actuelle est qu'il n'est pas possible de définir plusieurs "niveaux" (appelés aussi contexte) de vol correspondant au cas d'un algorithme récursif... Le voleur et la victime coopèrent pour le traitement des requêtes de vol.

Sachant que ces deux types de vols (concurrent: Kaapi ; coopératif: X-Kaapi) ont leurs avantages/désavantages en fonction de la classe de problème, la question naturelle qui se pose est :

Q 4. comment coupler efficacement un vol concurrent avec un vol coopératif ?

De plus, "qui peut le plus, peut le moins", la question préliminaire est :

Q 5. faut-il conserver un vol coopératif ? Sachant qu'il induit une très forte dégradation des performances dans le cas d'algorithmes à gros grain et que le vol concurrent possède un surcoût raisonnable.

Enfin pour terminer cette section, nous avons observé que nous avons de bien meilleurs performances si les données des algorithmes parallèles sont distribués sur les différents bancs mémoire.

Q 6. quel description hiérarchique de la machine faut-il donner ?

Une description par domaine de partage mémoire hiérarchique tel qu'utilisé dans X-Kaapi (mais hard-codé) ou telle que sortie par les outils libtopology (??) semble naturelle. De plus il est simple d'associer une certaine sémantique pour les déplacements mémoire : pas de contrôle évident pour les caches si ce n'est par localité temporelle ; contrôle possible sur les bancs des architectures NUMA ; contrôle de placement sur les architectures à mémoire distribuée.

Une sous question que l'on doit se poser et qui dépend aussi des algorithmes de la question 5 :

Q 7. faut-il voir la topologie du réseau autrement que de manière hiérarchique ? i.e. doit-on voir la topologie du réseau hypertransport d'idkoiff ?

Q 8. quel algorithme de vol permet d'exploiter au mieu cette hiérarchie ? A Jean-Noël de nous donner la réponse théorique et expérimentale : les choix fait dans Kaapi et X-Kaapi de toujours favoriser un vol local avant un vol distant sont à justifier et à améliorer.

1.2.3 Communication inter "memory domain"

Enfin pour terminer, il convient de voir comment traiter les aspects communications inter domaine de partage. Le modèle d'architecture dans X-Kaapi est actuellement le suivant :

- une hiérarchie de "domaines de mémoire" correspondant à la hiérarchie mémoire que l'on retrouve sur les architectures actuelles: cache L1 $\Rightarrow$ cache L2 $\Rightarrow$ cache L3 $\Rightarrow$ banc mémoire $\Rightarrow$ mémoire partagée $\Rightarrow$ mémoire distribuée. Certains niveaux faisant apparaître les organisations en cluster ou grid pourrait être ajoutés.

- un ensemble de (K-)processeurs associé à chaque niveau : pour un niveau donné, les processeurs associés à un niveau peuventt communiquer en utilisant la mémoire de ce niveau.

Une approche similaire a été choisi pour la définition des clusters dans Kaapi, sans descendre au niveau local (mémoire partagée et en dessous).

Pour chaque niveau, il existe une certaine manière de (liste non exhaustive, par exemple ajout d'un broadcast ?) :

- lire une donnée,
- écrire une donnée.

Outre cet aspect transport de données, les threads ont aussi besoin de se synchroniser.

Dans X-Kaapi, la communication & synchronisation entre 2 thread est gérée par une zone tampon (buffer) associée à un état. Seul l'écriture à distance est autorisée (cf papier renpar). Un thread voulant envoyer une donnée à un autre doit :

- écrire dans la zone tampon du destinataire,
- changer l'état associé.

A la charge du destinataire de vérifier le changement d'état. Si ce coût de scrutation est important il convient d'ajouter la capacité à exécuter un traitement (interruption chez le destinataire).

Dans Kaapi la communication est gérée 1/ soit à travers la mémoire partagée 2/ soit par message actif.

Il s'agit ici de se poser la question d'une interface uniforme pour la communication entre threads. Il me semble qu'il faille s'orienter vers une interface de type écriture à distance + signalisation et ma dernière question :

Q 9. doit-on chercher à utiliser une bibliothèque externe de communication ?

Q 10. quelle interface de communication ?

Chapter 2

API specification

2.1 Architecture model

The target architecture for X-Kaapi is a grid of clusters of multi-core, many core nodes. The architecture is assumed to hierarchically organized. At each level, all the cores use the same way to communicate together. The architecture is viewed as a non uniform memory architecture. At the finer level of the hierarchy, each core has a private memory (L1 cache). The second level of cache (L2), if exists, may be shared or not depending of the processor family / vendor.

For each node, the main memory is assumed to be organized by memory banks with fast access for the core(s)'s owner. Other cores have an access through a high performance network (*e.g.* idkoiff memory organization).

It is to the responsibility of the communication API (section **??**) to virtualize the memory organization in order to offer a common way to communicate. Nevertheless, the implementation should take care of the architecture to used to best method available.

2.2 Execution model

A program is composed of several threads of control into several process. Each process begins to execute the main entry point of the associated binary[1]. Each process is multithreaded and only the main thread executes the initialization code before returning to the main user code. All processes have an unique identifier.

The initialization will create several worker threads that are called k-processors, and one k-processor pursues the execution of main entry point of the process. It is called the main k-processor. All other k-processors begins by stealing work.

Each k-processor has two kinds of memory:

- A private memory that only the owner k-processor may access. This memory is associated to the C stack of function calls used by local computation done by the k-processor.
- A write only shared memory that all k-processors could write but not read. This memory is used to put data structure shared between k-processors to implement the work stealing operations.
- A shared memory that all other k-processors could read but not read. This memory allows read and write operations by all the k-processors and may be used to communicate user level data.

[1]It possible to have multiple binary, even if a SPMD model simply the complexity of the compilation / deployment stage.

All the following API functions are designed such that communication due to the work stealing algorithm between threads use the write only shared memory in order to do remote write operation: a k-processor never read remote data during the work stealing operations.

Note: promotion between these kinds of memory have to be discussed.

2.2.1 Memory operations ordering API

We assumed that the memory read and write operation may be reordered by the compiler or the hardware. To enforce an ordering constraint on memory barrier we assume the availability of *memory barrier*. Due to the unknown memory models of futurs target architectures of X-Kaapi, we assume two instructions to enforce reordering:

`kaapi_write_membarrier()` : ensure that all write operations prior to the barrier will have been committed to memory prior to any write following the barrier.

`kaapi_read_membarrier()` : ensure that all read operations prior to the barrier will have been committed to memory prior to any read following the barrier.

Depending of the architecture, implementation of the functions may enforce a stronger ordering constraint that strictly required.

2.3 Topology API

People involved:

The purpose of this API is to offers a view of the architecture. Libtopology[2] seems to be a complete API[3].

2.4 Workstealing API

People involved:

This section describes all the functions required to create stealcontext structure and managing action at the different points of the work stealing algorithm.

2.4.1 k-processor management

Add the begining of the computation a certain number of k-processors are automatically created: the number of k-processors and their mapping on physical processors are controlled by environment variables defined in section **??**.

2.4.2 Adding or deleting k-processors

Nevertheless, it is possible to dynamically adjust the number of running k-processors of a process by setting the *concurrency* of X-Kaapi.

[2]Renamed hwloc for hardware locality: http://www.open-mpi.org/projects/hwloc/

[3]The effectiveness use (simplicity, completeness of functions) of hwloc should be evaluated.

int kaapi_set_concurrency(int concurrency) : change the current number of running k-processors to the new value $concurrency$. $concurrency$ is a non negative or null integer. If the old value is less or equal than the new value, then additional $(new\_value - old\_value)$ k-processors are created. Else $(old\_value - new\_value)$ k-processors are deleted.

Newly added k-processor begins to steal work from other k-processor. The k-processor is mapped onto the next free physical resource. This operation does not required any kind of synchronization.

Work or results owned by a deleted k-processor are attached to the closest alive k-processor following the hierarchy in order to minimize communication cost. In this case synchronization between deleted k-processor and the newly attached k-processor have a synchronization.

Return value is 0 (KAAPI_OK) in case of success else one of the following error code is returned:

KAAPI_EINVAL : invalid argument, typically passing 0 or a negative value.

KAAPI_EGAIN : the system lacked the necessary resources to create another k-processor.

---

int kaapi_get_concurrency(void) : return the current number of running k-processors.

---

kaapi_processor_t* kaapi_self_processor(void) : return a pointer to the current running k-processor.

2.4.3 Posting a request

---

int kaapi_request_init(kaapi_request_t* kpr : initialize a new request data structure. This function must be called before any use of the request data.

Return value is 0 (`KAAPI_OK`) in case of success else one of the following error code is returned:

KAAPI_EINVAL : invalid argument, typically passing 0 or invalid pointer.

---

int kaapi_request_destroy(kaapi_request_t* kpr : destroy an already initialized request data structure. After the call to the function, the request data cannot be used in any function.

Return value is 0 (`KAAPI_OK`) in case of success else one of the following error code is returned:

KAAPI_EINVAL : invalid argument, typically passing 0 or invalid pointer.

---

int kaapi_request_post(kaapi_processor_t* thief kaapi_processor_t* victim, kaapi_request_t* kpsr): the thief running k-processor post a steal request to the victim processor `victim`. This is a non blocking call: in case of success, the running thief k-processor has to periodically check (`kaapi_request_test`) or wait (`kaapi_request_wait`) for the completion of the request.

Return value is 0 (KAAPI_OK) in case of success else one of the following error code is returned:

KAAPI_EINVAL : invalid argument, typically passing 0 or invalid pointer.

KAAPI_EBUSY : the victim processor does not accept request.

---

int kaapi_request_test(kaapi_request_t* kpsr): return 0 if the request has been processed, else return one of the follwing error code. On successful return, the status a call to kaapi_request_status allows to return the status the request.

KAAPI_EINVAL : invalid argument, typically passing 0 or invalid pointer.

KAAPI_EINPROGRESS : the request is not yet processed.

KAAPI_EINTR : the processing of request has been interrupt, may due terminaison.

---

int kaapi_request_wait(kaapi_request_t* kpsr): return 0 if the request has been processed, else return one of the follwing error code. On successful return, the status a call to kaapi_request_status allows to return the status the request.

KAAPI_EINVAL : invalid argument, typically passing 0 or invalid pointer.

KAAPI_EINTR : the processing of request has been interrupt, may due terminaison.

---

int kaapi_request_status(kaapi_request_t* kpsr): return the status of the request using the following code

KAAPI_OK : the request has successful steal work.

KAAPI_EAGAIN : the request does not steal work and it could be reposted to other k-processor.

KAAPI_EINTR : the processing of request has been interrupt, may due terminaison.

KAAPI_EINVAL : invalid argument, typically passing 0 or invalid pointer.

---

int kaapi_request_cancel(kaapi_request_t* kpsr): cancel a request posted to a k-processor. If successful, the kaapi_request_cancel function will return zero. Otherwise, an error number will be returned to indicate the error.

Implementation note

This function is generally difficult to implement. The victim processor may have already stored the reply while the thief is trying to cancel the request. If the protocol to ensure consistency required a strong synchronization in every reply, then this function may be suppressed from the API.

KAAPI_OK : the request has been successfully canceled.

KAAPI_EBUSY : the request has been already replied with a result.
A call to kaapi_request_status function will return its status .

KAAPI_EINVAL : invalid argument, typically passing 0 or invalid pointer.

2.4.4 Access to user defined arguments of a request

A request data structure contains a buffer of `KAAPI_REQUEST_DATA_SIZE` bytes (at least 16 bytes) that could be used to store user defined arguments. The data structure `kaapi_request_t` contains an opaque field that points to the first byte of this buffer.

Implementation note

Does the size of buffer should be bigger ? Does the buffer size should dynamic ?

In order to read or write this data, the following functions should be used:

`int kaapi_request_writedata(kaapi_request_t* kpr, int count, const void* src)`: write `count` bytes from `src` to the internal request buffer. Upon successful completion the number of bytes which were written is returned (non negative integer). If a negative integer is return then the absolute value indicate the error:

KAAPI_EINVAL : invalid argument, typically passing 0 or a negative value.

KAAPI_ENOMEM : the request lacked the necessary resources to store count byte.

`int kaapi_request_readdata(kaapi_request_t* kpr, int count, void* dest)`: read `count` bytes from the internal request buffer to `kpr`. Upon successful completion the number of bytes which were read is returned (non negative integer). If a negative integer is return then the absolute value indicate the error:

KAAPI_EINVAL : invalid argument, typically passing 0 or a negative value.

KAAPI_ENOMEM : the request lacked the necessary resources to store count byte.

`const void* kaapi_request_data(kaapi_request_t* kpr)` This function may only by called by the owner of the request (the thread that initialized it). Upon successful completion a pointer to the first byte of the internal buffer is returned. Else it returns 0.

0 : invalid argument, typically passing 0 or a negative value, or the request pointer is not owned by the caller

Implementation note

Testing if the request is owned by the caller thread may required extra informations that are not required for normal execution. This extra informations/code should be available during debugging compilation mode. Thus the value retuned by the function may has to be defined as 'undefined' is the call does not occur with the context of the owner...

2.4.5 Selection of victim k-processor

2.5 StealContext

Pushing and poping a StealContext structure

Cooperative processing of steal request

Concurrent processing of steal request

Finalization point

Preemption and preemption point

StealContext data structure

2.6 Implementation

In order to avoid most of dynamic memory management, the maximal set of k-processors per process is bounded by a constant `KAAPI_MAX_KPROCESSORS` which at least the number of physical cores on the machine. Each k-processor has an unique local (process wide) identifier in $[0, KAAPI\_MAX\_KPROCESSORS - 1]$.

2.6.1 `kaapi_processor_t` data structure

Each k-processors have a `KAAPI_MAX_KPROCESSORS` array of requests that posted by other k-processors. If the entry i is non null, then the k-processor with local identifier i has posted a request. Moreover, each k-processor has a flag indicating if at least one request has been posted. The structure `kaapi_processor_t` has at least the following definition of fields:

```
struct kaapi_processor_t {
  unsigned int     _localid;
  kaapi_request_t* _request[KAAPI_MAX_KPROCESSORS];
  int              _flag; /* 0 or 1 */
};
```

2.6.2 `kaapi_request_t` data structure

2.6.3 Algorithm to post a request

The algorithm to post a request is the following[4]:

```
1.  int kaapi request post(
2.          kaapi processor t* thief,
3.          kaapi processor t* victim,
4.          kaapi request t* kpr )
5.  {
6.    kpr->_state = KAAPI_REQUEST_POSTED ;
7.    kaapi_write_membarrier();
8.    victim->_request[ thief->_localid ] = kpr ;
9.    victim->_flag = 1;
10. }
```

[4]Not that the current implementation is not this one: in place of set to 1 the flag on the victim, an atomic increment is executed on the flag.

Line 7 is required memory barrier to ensure that previous write operations will be view prior to write operations following the barrier : If the victim k-processor views the request (line 8 committed to the memory), then the request data will also be viewed in a coherent state.

2.6.4 Management of data

2.7 Threads and synchronizations API

People involved:

2.7.1 Thread creation

2.7.2 Mutex

2.7.3 Condition

2.8 Communication API

People involved:

2.9 Parallel programming API

People involved:

2.10 Deployment of program

2.10.1 Environment variables

2.10.2 Extension of the library