Specification X-Kaapi

V. Danjean, T. Gautier, C. Laferrière, F. Le Mentec

November 12, 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

Overview of the API

In this chapter, we presents an overview of the API. Next chapter will presents the specification for each function of the API, especially the functions that deal with posting and waiting for requests, accessing to the topology, that are not strictly required for the standard developer of parallel algorithm with X-Kaapi.

2.1 Introduction to the task model

In X-Kaapi, we assume that a parallel program is composed by a set of concurrent threads of control. During the initialization, a certain number of threads are created, and one of them is called the *main thread*. This threads may be organized in set of processes (UNIX process) and in this case they can directly communicate through a shared memory. Threads that does not share memory should use inter-process communication primitive (provided by the runtime). Each thread has a *global identifier* which is unique for all the execution of a parallel program.

Each thread is structured as a ordered sequence of tasks. The task is the basic unit of execution: it may be executed, stolen, stopped and preempted. The first part of the API only provide this task model, with different kinds of tasks.

independent : a non executed task may be stolen without any consideration about dependency. This implies that a thread that try to execute a stolen *independent task* can pass to the next task. The requirement to define an independent task is only to give the entrypoint to do the computation.

adaptive : an *adaptive* task should provide itself a method to steal work, and this is the only requirement.

data flow : this the Kaapi/Athapascan task with data flow dependencies. The user provides the way of doing computation and the runtime offers a way to steal it. The data flow dependencies are managed with a lighter version of the Kaapi/Athapascan data flow graph representation.

This different kinds of tasks implies different requirements in the synchronization: independent tasks only required that all the tasks have been executed after a synchronization barrier (`a1::Sync` or Cilk sync) or at the end of the program. Using adaptive tasks to program adaptive programs, the main sequential task may want to preempt or wait the end of the theft tasks. The most efficient implementation of the execution of data flow tasks in old Kaapi/Athapascan was to execute them using their sequential order of creation until the thread of control wants to execute a stolen task. Them it suspends its execution and tries to steal other tasks from other threads.

2.1.1 Target architecture

This task model aims to be portable on different machines:

Multicore : this is the first target architecture, mostly due to development was done on such machine. On this kind of machine the target operating system is Linux and Mac OS X[1] Other operating system may have to be considered but are not in the main goal of releasing a first version (AIX, SunOS).

GPU : because GPU contains several (up to 30) multiprocessors, each multiprocessors contains several stream processors, it is possible to port workstealing strategy inside the GPU. In a first approximation, I will only see on main technical problem that remains to solve: because in current CUDA version only one kernel is running at any time on the card, we should provide a way to generate of a big-kernel that is able to call all entrypoints of the tasks. The second problem is due to the fact that each multiprocessor in NVidia card is efficiently used using numerous independent threads: classical GPU program generates a lot of independent threads that run concurrently on the cores; thus it should be able at runtime to reorganize the use of multiprocessors: a task (= numerous CUDA threads) may be stolen by one multiprocessor or may have interest in using several multiprocessors.

MPSoC : due to our collaboration with ST micro-electronics. On this borad we have currently access only to OS21 operating system. It offers the notion of task that corresponds to a thread of control with a cooperative schedule or a preemptive schedule and offers synchronization primitives as well as communication between several cores of the chip.

2.1.2 Memory model

As for most of the modern parallel language (Titanium, UPC, X10, Fortress) it is important to provide referential transparency to the user data in order to simply parallel programming of algorithm.

The physical memory is hierarchical: several cache levels between cores and the shared memory (non-uniform access) of multiprocessor; memory of the multiprocessors inside a cluster, etc...

The main basic assumption made by X-Kaapi is to that: 1/ if a sequential code pass user data to the library, then it can only be read after the parallel computation or at certain synchronization points; between sequential phase, this user data will contains intermediate versions or partial version of the computation. 2/ user data will never by freed by the library.

2.2 KAAPI API levels

X-Kaapi will be organized in an API composed of several modules (currently at least 3):

Module 0 : written in C it should provide the task management and scheduling on top of multi-cores machines as well as the MPSoC and GPU[2].

Module 1 : a subset of POSIX thread API. Because multithreaded computation in X-Kaapi will be mixed with other libraries and programs, it may be necessary to provide a strong compatibility (in term of guaranteed performances ?) when someone wants to mix X-Kaapi with other codes.

[1] No choice: I develop on MacOS X.

[2] in the futur....

Module 2 : the Athapascan C++ interface reused for compatibility with historic codes. The Athapascan C++ interface will be extended in order to be able to explicitly pass the current thread that creates objects (tasks and shared objects).

Module 3 : In this module a new interface will have to be developed in order to facilitate the re-engineering of numerical code that use a lot of the array data structure (...). This module has to be entirely design and developed.

2.3 Task model

A parallel X-Kaapi program is composed by a sequence of tasks. Tasks are pushed into a stack. A task may creates several tasks. The order of creation follows the *reference order* proposed few years ago in Athapascan/Kaapi: When a thread finishes to execute a task, it begins to execute the children tasks of the executed task, following their order of creation (the first created task is the first executed task).

A task is a constant size structure with one parameter and several attributs[3].

A task as A task is composed of the following data:

flags : a bit fields used to store various informations about the task.

a body : a C function used to execute the instructions of a task.

Optional parameter for adaptive task.

a splitter : a C function used to execute the extract work of a task.

If not defined a default splitter function will be called to extract work of a task.

a stack pointer : that points to the entry in the stack where are stored the parameters.

See section 2.3.3 how to pass parameters to a task. Note that because of the order of execution of X-Kaapi program which required to create all tasks of a running task before executing them, the stack pointer for each task's parameters should be stored in order to recover the parameters of each task.

2.3.1 Stack of tasks

The creation of a task is done by storing the data that defined a task into a *stack*. A stack contains created tasks. A stack could also store data. In fact the X-Kaapi stack should be viewed as two stacks a stack to store tasks and a stack to store data. The current execution state of stack is defined by three fields (see figure 2.1).

pc : the program counter that points to the next task to execute.

sp : the stack pointer that points to the next task to push into the stack.

sp_data : the data stack pointer that points to the next memory where to store data.

Each of the tasks and data stacks is represented by a contiguous memory (an array).

They are two reasons to not use a C-stack for storing tasks or data:

[3]In a previous version of this document, I was more interested in providing a fixed set of parameters to a tasks that allows fastest execution. But it also will make too complex the implementation of the C++ interface because some parameters will be stored into the task data structure will other will be on the stack.

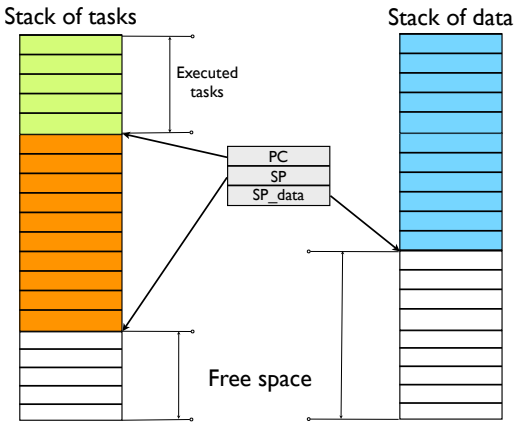

Figure 2.1: X-Kaapi stack: a dual stacks structure. At the left part of the figure, the stack of tasks with both `PC` (program counter) and `SP` pointer (stack pointer). At the right, the stack of data.

- when the runtime does not execute the body of a task, the C-stack could not be saved. Which allows to port X-Kaapi on top of architecture without considering the way to save / restore processor context.
- thus it also permits to optimize context switch on most situation if the only point in the control flow where a thread waits are between tasks.

The algorithm 2.2 to execute all tasks of a stack does not required recursion between task's bodies (it is a little bit more efficient if at the end of a task, we directly execute children tasks).

2.3.2 Task body

The task body is a C-function that takes only two parameters:

kaapi_task_t* : a pointer to the task to execute which also contains the stack pointer that points to the list of parameters.

kaapi_stack_t* : the stack to store child tasks created during execution of the task body.

When a task body returns, the X-Kaapi runtime test is child tasks have been created. If it is the case, the runtime enqueue a special task (unstealable) which will restore the context of the stack before execution the task body[4].

2.3.3 Passing arguments

The parameters of a task are pushed into the X-Kaapi stack. A X-Kaapi stack may allocate data using `kaapi_stack_pushdata(stack, size)`. If the call return a non null pointer then at least size bytes of memory are allocated on the stack. Several allocation may be done on the stack. The task ' stack pointer must be initialized to the stack pointer return by a call to `kaapi_stack_pushdata`.

[4]This task is very similar to blr PowerPC instruction or ret/retn instruction in the X86 instructions set.

```
redo_work:
  {
    if (task->body ==0) return 0;

    /* save stack pointers, pc is equal to task ! */
    saved_sp      = stack->sp;
    saved_sp_data = stack->sp_data;
    (*task->body)(task, stack);
    task->body = 0;

    /* push restore_frame task if newly created tasks */
    if (saved_sp < stack->sp)
    { /* push retn task that will restore stack pointers
         before executing the task 'task' */
      retn = kaapi_stack_top(stack);
      retn->body  = &kaapi_retn_body;

      /* store the saved frame in parameters */
      retn->sp = kaapi_stack_pushdata(stack, sizeof(kaapi_frame_t));
      frame = (kaapi_frame_t*)retn->sp;
      frame->pc           = task;   /* <=> save pc */
      frame->sp           = saved_sp;
      frame->sp_data = saved_sp_data;

      /* commit the task to the stack */
      kaapi_stack_push(stack);

      /* update pc to the first forked task */
      task = stack->pc = saved_sp;
      goto redo_work;
    }

    task = ++stack->pc;
    goto redo_work;
  }

  task = ++stack->pc;
  if (stack->pc >= stack->sp) return 0;
  goto redo_work;
}
```

Figure 2.2: Algorithm to execute tasks of a stack.

2.3.4 Shared objects

A shared object allows to define a set of writers and readers that are synchronized through the produced value. Two tasks that share a value through a shared object have been created together from a unique point in the past of the computation. This shared object is the classical shared object in Athapascan/Kaapi.

In order to extend the kind of coordination between tasks that is permitted with X-Kaapi, we introduce a fifo shared object. Such object could be *anonymous* and after its creation both the producers and consumers have to be forked with the fifo shared object as parameter. Producers and consumers are not synchronized by a read / write dependency but by the availability of data into the fifo shared object. A consumer could only be started iff a value has been produced into the fifo shared object.

Nevertheless, we also introduce a *named* fifo shared that could be defined by parallel tasks: two tasks without a priori any relationship could defined the same fifo shared object given the same *name*. X-Kaapi will manage the localisation of the producers and concumers in order to transport data from a task to an other.

2.3.5 Access mode

Each type should declare the access mode and type of each of its parameters. The access modes are the same than in Athapascan/Kaapi. Please refer to documentation about Athapascan to have a complete description of the semantic.

v : the parameter is passed by value. A copy was made and the task may modify the copy without change the value of the effective parameter.

r : shared read access. The task may read the value of the shared object.

rp : shared read postponed access. The task may create a task that will read the value, but it cannot read itself the value.

w : shared write access. The task may write the value of the shared object. If the task do not write a value, then an undefined value is write to the shared value.

wp : shared write postponed access. The task may create a task that will write the value, but it cannot write itself the value.

rw : shared exclusive access. The task may read and write the value of the shared object.

rwp : shared exclusive postponed access. The task may create a task that will read or write the value, but it cannot read or write itself the value.

cw : shared concurrent write access. The task may accumulate a value of the shared object. The final value depend on the number of created task with concurrent write access.

cwp : shared concurrent write postponed access. The task may create a task that will accumulate a value, but it cannot accumulate itself a value.

E : is a special coding for empty mode, either because the access mode is unknown or has not yet be set.

2.3.6 Fibonacci example

Figure 2.3 presents the code for Fibonacci example using the stupid recursive code:

$$fibo(n) = \begin{cases} n & \text{if } n < 2 \\ fibo(n-1) + fibo(n-2) & \text{else} \end{cases}$$

The program given in figure 2.3 is based on creation of tasks for each recursive calls as well as for the continuation to sum the produced value. This summation has data flow dependencies between the both previous forked task. The access modes are defined by explicitly

```
/* Fibonacci args of a task body */
typedef struct arg_fibo { int n; int* res; } arg_fibo;

/* Fibonacci task body */
void fibo_body( kaapi_task_t* task, kaapi_stack_t* stack )
{
  arg_fibo* arg = (arg_fibo*)task->sp;
  if (arg->n < 2)
  { /* store the input 'n' to the output value */
    *arg->res = arg->n;
  } else {
    int* result1 = (int*)kaapi_stack_pushdata(stack, sizeof(int));
    int* result2 = (int*)kaapi_stack_pushdata(stack, sizeof(int));

    kaapi_task_t* task1 = kaapi_stack_top(stack);
    task1->flags     = KAAPI_TASK_DFG;
    task1->body      = &fibo_body;
    arg_fibo* a1 = (arg_fibo*)kaapi_stack_pushdata( stack, sizeof(arg_fibo) );
    task1->sp = a1;      a1->n = arg->n-1;      a2->res = result1;
    kaapi_stack_push(stack);

    kaapi_task_t* task2 = kaapi_stack_top(stack);
    task2->flags     = KAAPI_TASK_DFG;
    task2->body      = &fibo_body;
    arg_fibo* a2 = (arg_fibo*)kaapi_stack_pushdata( stack, sizeof(arg_fibo) );
    task2->sp = a2;      a2->n = arg->n-2;      a2->res = result2;
    kaapi_stack_push(stack);

    kaapi_task_t* task_sum = kaapi_stack_top(stack);
    task_sum->state    = KAAPI_TASK_DFG;
    task_sum->body     = &sum_body;
    arg_sum* a3 = kaapi_stack_pushdata( stack, sizeof(arg_sum) );
    task_sum->sp = a3; a3->res = arg->res;
    a3->result1 = a1->res; a3->result2 = a2->res;
    kaapi_stack_push(stack);
 }
}
```

Figure 2.3: Classical Fibonacci example

register the format of the task. The figures 2.4 and **??** present a detailed of the content of the X-Kaapi stack.

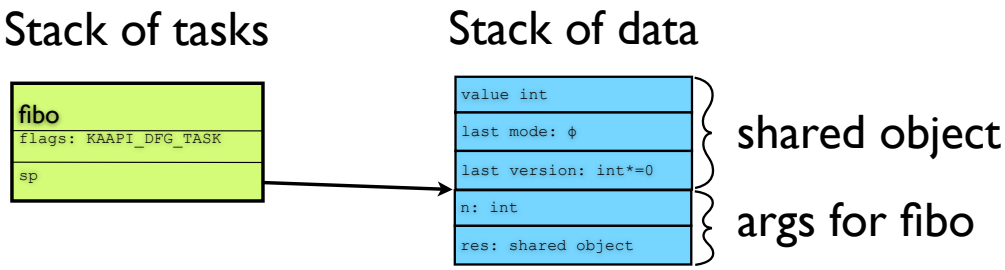

Figure 2.4: X-Kaapi stack before executing one instance of Fibonacci task. Task is pushed into the stack of tasks and the global data is pushed into the stack of data of the same X-Kaapi stack. Such construction is done during each recursive creation of Fibonacci task in the algorithm in figure 2.3.

2.3.7 Data flow graph computation

The purpose of this section is to present the new algorithm used in X-Kaapi to compute data flow constraints: *i.e* how to compute the fact that a data is ready to be used in as parameter of task.

Some propositions are naturally deduced from the access mode and the semantic of Athapascan API.

Proposition 1 *Let us consider a task t with k parameters. The following propositions are true:*

1. *A task without parameters is always ready to execute.*
2. *A effective parameter with access mode **[v]** is always ready to execute the task t.*
3. *A effective parameter with access mode **[w]** or **[wp]** is always ready to execute the task t.*
4. *A effective parameter with access mode **[r]** or **[rp]** is ready to execute the task t iff the previous access(es) that write (**[w]**, **[wp]**, **[rw]**, **[rwp]**, **[cw]** or **[cwp]**) the value are parameters of tasks that have been already executed.*

The following proposition also holds:

Proposition 2 *The first task of a stack is always ready except iff it is not stolen.*

Sequential execution of a X-Kaapi program, could be view as a repeated use of proposition 2: the first task is created in state *not stolen*, then it is ready. Thus it is executed and poped. The next task becomes the first of the stack.

Proposition 3 *The sequential execution of a X-Kaapi program is to repeatedly execute the next task pointed by `pc` of the stack attached to the current flow of control using the algorithm 2.2.*
No data flow constraints need to be computed.

2.3.7.1. Representation of objets

An access is an effective parameter of a task to a shared object. An access to a shared object is a pointer to the global data and an optional range interval of values. The global data stores a data, and the last version of the data and its access mode; these fields are only used by the work stealing algorithm to detect concurrent access. The figure 2.5 shows definitions of access and shared object. In previous version of Kaapi, all accesses to the same shared object are linked together following the order of creation of the task. This chain of accesses was

```
struct kaapi_access_t  {
  kaapi_gd_t* gd;     /* global data that store value to use during
                         the execution of the task */
  /* optional value */
};

struct kaapi_access_steal_t  {
  void* data;         /* data to use during the execution of the task */
  /* optional value */
};

struct kaapi_gd_t {
  void*         data;         /* data used during normal execution */
  kaapi_amode_t last_mode;    /* access mode of the last version */
  void*         last_version; /* last version, used during work stealing */
};
```

Figure 2.5: Definition of access and shared object data structures

used to computed the readiness property of tasks. In the next sections, we propose an other algorithm that does not need to chain together accesses: it will improve both the memory usage of the graph representation as well as the overhead of T_1 versus T_s. The complexity of the algorithm that runs on each steal request remains equivalent.

2.3.7.2. Execution by workstealing
The work stealing algorithm will change the sequential execution: a thief may has stolen (a ready) task that will produce results read by next tasks in the execution order. At the instant of the work stealing decision to steal a task, the next tasks are probably not known. In order to avoid the computation of the data flow constraints before execution of each task, like in the first version of Athapascan, X-Kaapi is based on the following strategy:

1. The stolen task in the stack is marked as 'stolen'.
2. The algorithm 2.2 executes tasks until the next task to execute is marked 'stolen'.

Thus next tasks of a stolen task may have dependencies, the sequential execution is suspended and the thread begins to steal task from other threads, eventually from itself.

Using such strategy, the data flow constraints are only computed during work stealing operations that are, in theory, not very many. This strategy was already implemented in Kaapi, the previous version of X-Kaapi.

2.3.7.3. Computation of data flow constraints during work stealing operation
The most important difference between Kaapi and X-Kaapi is that, in X-Kaapi the data flow graph representation is lighter: accesses to a shared object is not linked together; tasks in the stack are not linked together. In order to detect ready task, the selection algorithm during work stealing operation uses `last_XX` data fields of the `kaapi_gd_t` data structure (see above, figure 2.5) to compute *version* of a chain of accesses: the selection algorithm iterates through all tasks in the stack, and then try to detect if all accesses of task is ready (and then the task will be ready to be stolen). The detection of readiness property of an access is done using the following rules:

- if the task is not executed (candidate task): let us note by `m` the access mode of the considered access and by `(last_mode, last_version)` the corresponding data fields of the associated `kaapi_gd_t` data structure.

- if m is a postponed mode, then the access is ready. Then set `last_mode` to `m`.
- if m is a write mode (`w` or `cw`) then the access is ready. Then set `last_mode` to `m` and `last_version` to `0`.
- if m is a read mode (`r`) then the access is ready iff `last_mode == E` or `last_mode == r` and `last_version !=0` (the value has been produced). Then set `last_mode` to `r`. The field `last_version` represent the data that will be read by the task.
- if m is a exclusive mode (`rw`) then the access is ready iff `last_mode == E` and `last_version !=0` (the value has been produced). Then set `last_mode` to `rw`. The field `last_version` represent the data that will be accessed by the task.

- if the task was terminated (already stolen and marked as terminated), then:
  - if m is a write access, then set `(last_mode,last_version)` to `(w, value_produced)`.
  - if m is a read access, then set `last_mode = r`
  - if m is a exclusive access, then set `(last_mode,last_version)` to `(rw, value_modified)`.

2.3.8 Examples

2.3.8.1. Adaptive Fibonacci example

```
void fibo_body( kaapi_stack_t* stack, int n, int* result )
{
  int result1;
  int result2;

  /* C nested function called during steal op */
  void splitter( int event, int count, kaapi_stack_t** thief_stacks )...

  if (n < 2) {
    *result = n;
  }
  else {

    /* push a task to reply to a steal request*/
    kaapi_task_t* task = kaapi_stack_top(stack);
    task->event = KAAPI_TASK_ADAPTIVE;
    task->body = (kaapi_task_body_t)&splitter;

    /* test if steal request */
    kaapi_stealpoint( stack, task, &splitter );

    /* else push the task */
    kaapi_stack_push( stack );

    /* recursive call */
    fibo_body( stack, n-2, &result2 );

    /* wait all thiefs (if any):
       return 0 if no thief has tries to steal the task */
    if (kaapi_finalize_steal(stack, task) == 0)
    {
      /* not theft, so do it myself  */
      fibo_body(stack, n-1, &result1 );
    } /* else wait */

    /* return with sum */
    *result = result1 + result2;
 }
}
```

Figure 2.6: Adaptive version of the Fibonacci example

```
void splitter( int event, int count, kaapi_stack_t** thief_stacks )
{
  int i;
  typedef struct arg_fibo {
    int n;
    int* res;
  } arg_fibo;

  /* entry point called to executed exported task */
  void entrypoint( kaapi_task_t* task, kaapi_stack_t* stack )
  {
    arg_fibo* a = (arg_fibo*)task->sp;
    fibo_body( stack, a->n, a->res );
  }

  /* search for the first thief */
  for (i=0; i<KAAPI_MAX_REQUEST; ++i)
    if (thief_stacks[i] !=0) break;
  kaapi_task_t* task = kaapi_stack_top(thief_stacks[i]);
  task->flags = KAAPI_TASK_INDEPENDEND;
  arg_fibo* a=(arg_fibo*)kaapi_stack_pushdata(thief_stacks[i],sizeof(arg_fibo));
  task->sp = a;
  a->n     = n-1;
  a->res   = &result2;
  task->body = (kaapi_task_body_t)&entrypoint;
  kaapi_stack_push( thief_stacks[i] );
}
```

Figure 2.7: Splitter function that export a task in case of steal request

2.3.8.2. Transform example

```
template<class InputIterator, class OutputIterator, class UnaryOperator>
void TransformStruct<InputIterator,OutputIterator,UnaryOperator>::
  doit(kaapi_stack_t* stack)
{
  if (stack ==0) stack = kaapi_self_stack();

  /* push a task to test and process steal request */
  kaapi_task_t* task = kaapi_stack_top( stack );
  task->flags    = KAAPI_TASK_ADAPTIVE;
  task->body = 0;

  /* local iterator for the nano loop */
  InputIterator nano_iend;

  /* amount of work per iteration of the nano loop */
  int unit_size = 512;
  int tmp_size = 0;

redo_work:
  while (_iend != _ibeg) {
    /* the steal point where splitter is called in case of steal request.
       -here _iend is pass as inout parameter and updated in case of steal */
    kaapi_stealpoint( &stack, &task, &splitter, _ibeg, _iend, _obeg, _op );

    tmp_size =  _iend-_ibeg;
    if (unit_size > tmp_size) { unit_size = tmp_size; nano_iend = _iend; }
    else nano_iend = _ibeg + unit_size;

    /* sequential computation: push task action in order to allows steal at
       this point while I'm doing seq computation */
    kaapi_task_push(stack, task, &splitter);

    _obeg = std::transform( _ibeg, nano_iend, _obeg, _op );

    /* return from sequential computation: pop task action in order to disable
       any steal at this point while I'm doing seq computation */
    kaapi_task_pop(stack);

    _ibeg += unit_size;
  }
  /* preempt thief */
  if (kaapi_preempt_next_thief( task, 0, (kaapi_reducer_function_t)&reducer, this )
    goto redo_work;

  /* definition of the finalization point where all stolen work a interrupt and col
  kaapi_finalize_steal( task );
}
```

Figure 2.8: Splitter function that export a task in case of steal request

2.3.8.3. Prefix example

2.4 Process startup and termination

The X-Kaapi library is initialized before the main entry point of process (*main* function). Options may be passed either by environment variable or by the command line arguments. When the control flow begins to execute the main function, all X-Kaapi options are deleted from the command line arguments[5].

Each process terminates either when the whole computation is terminated or because process has been killed.

[5]Je suppose qu'il doit êre possible de faire cela... bien que je ne sache toujours pas comment.

Chapter 3

API specification

3.1 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 3.8) 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.

3.2 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.

3.2.1 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.

3.3 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].

3.4 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.

3.4.1 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 3.10.1.

3.4.2 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.

3.4.3 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.

3.4.4 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...

3.4.5 Selection of victim k-processor

3.5 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

3.6 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 $[0, KAAPI\_MAX\_KPROCESSORS - 1]$.

3.6.1 `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 */
};
```

3.6.2 `kaapi_request_t` data structure

3.6.3 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.

3.6.4 Management of data

3.7 Threads and synchronizations API

People involved:

3.7.1 Thread creation

3.7.2 Mutex

3.7.3 Condition

3.8 Communication API

People involved:

3.9 Parallel programming API

People involved:

3.10 Deployment of program

3.10.1 Environment variables

3.10.2 Extension of the library