ChatGPT

Plan d'extension du parseur Hamon DSL en C++

Contexte et objectif de l'extension

Le projet **Hamon Cube** utilise un langage de configuration DSL personnalisé (fichiers `.hc`) pour décrire la topologie logique d'un mini-cluster hypercube et la distribution d'un job sur plusieurs nœuds. La version actuelle du parseur Hamon DSL est minimaliste et ne gère que quelques directives de base (nombre de nœuds, topologie, nœuds et rôles) [1]. Jusqu'ici, un fichier YAML statique (ex. `hamon.yaml`) décrivait la configuration des nœuds (IDs, voisins, rôles, CPU, NUMA, endpoints) [2] [3]. Le nouvel objectif est de remplacer ce YAML par le DSL Hamon enrichi, en étendant le parseur C++ pour prendre en charge de **nouvelles directives** et blocs de configuration de manière robuste. Il s'agit de permettre de décrire non seulement la topologie des nœuds, mais aussi les *jobs* à exécuter (avec phases, entrées, etc.), des réglages globaux (métriques, log…), des variables, et des raccourcis de sélection, le tout sans dépendances externes et avec une gestion rigoureuse des erreurs. L'API résultante doit fournir au programme orchestrateur toutes les informations nécessaires pour lancer les processus ou conteneurs de chaque nœud et orchestrer les phases du job de façon reproductible.

Nouvelles directives du DSL à supporter

Pour répondre aux besoins identifiés, plusieurs catégories de directives sont ajoutées au DSL : des directives de **topologie**, des blocs de **nœud**, des blocs de **job**, ainsi que des directives **globales** supplémentaires. De plus, le DSL prendra en charge des *raccourcis de sélection* pour référencer des ensembles de nœuds plus simplement, et introduira la notion de *variables* et d'expressions pour calculer certains paramètres dynamiquement. Voici un récapitulatif des nouvelles fonctionnalités prévues :

- **Directives de topologie** – Définissent la configuration globale du cluster :
- `@use <N>` : nombre total de nœuds logiques à utiliser (équivalent de la taille du cluster). Exemple : `@use 8` pour un cube à 8 nœuds [4].
- `@dim <d>` : dimension du hypercube (optionnelle, calculée automatiquement si non fournie et si `@use` est une puissance de deux [5]). Par exemple, `@dim 3` pour un hypercube 2^3.
- `@topology <type>` : type de topologie réseau (par ex. `hypercube` ou `ring`). *Par défaut*, la topologie est hypercube [6]. Si on choisit `hypercube`, le parseur validera que le nombre de nœuds est une puissance de deux ou qu'une dimension explicite est donnée [7]. S'il manque la dimension, elle sera déduite automatiquement comme log2(N) [7].

- `@autoprefix <HOST:PORT>` : préfixe d'adresse réseau à appliquer automatiquement aux nœuds. C'est une évolution de la directive `@auto` existante [8]. Cette directive indique un hôte de base et un port de base pour les endpoints TCP de chaque nœud. Le parseur stockera le hostname et le port de base, puis lors de la finalisation il attribuera à chaque nœud une adresse en incrémentant le port par l'ID du nœud [9] (par exemple, avec `@autoprefix 127.0.0.1:8000`, le nœud 0 aura `127.0.0.1:8000`, le nœud 1 `127.0.0.1:8001`, etc., sauf si un nœud a un endpoint spécifique override). Cela évite de renseigner manuellement chaque adresse de nœud.

- **Blocs de nœud** – Définissent les propriétés de chaque nœud logique du cluster. La syntaxe reste `@node <id>` pour ouvrir un bloc décrivant le nœud d'identifiant donné. À la suite, on attend

des directives imbriquées (indentées) spécifiant les attributs du nœud. Les nouvelles sous-directives prises en charge sont :

- `@role <type>` : rôle du nœud, par exemple `coordinator` (coordinateur) ou `worker` (travailleur), ou un rôle personnalisé sous la forme `custom:Nom` [10]. Si aucun rôle n'est spécifié pour un nœud, le parseur appliquera les *défauts* existants : le nœud 0 devient coordinateur par défaut, les autres en `worker` [11] [12].
- `@cpu numa=<I> core=<J>` : affectation aux ressources CPU/NUMA. Permet d'indiquer explicitement sur quel nœud NUMA et quel cœur CPU exécuter le processus du nœud logique [10]. Si omis, on utilisera les valeurs par défaut déjà en place (par exemple NUMA=0 et core = id du nœud) [12], ce qui correspond souvent à épingler chaque nœud logique sur un cœur distinct, avec les premiers nœuds sur le premier socket NUMA, etc.
- `@ip <HOST:PORT>` : définit explicitement l'endpoint réseau du nœud (adresse IP:port). Cela permet d'outrepasser l'autoprefix par défaut pour un nœud spécifique [13]. Le parseur utilisera une fonction utilitaire pour découper la chaîne en host et port (gérant IPv6 entre crochets, etc.) comme dans la version existante [14] [15]. Si seulement l'IP est fournie sans port, on appliquera un port par défaut (par autoprefix ou 8000+id) lors de la finalisation [9].
- `@neighbors [a,b,c]` : liste explicite des voisins logiques de ce nœud (IDs des nœuds connectés directement) [13]. Si aucune liste n'est fournie pour un nœud, le comportement dépend de la topologie : pour un hypercube, le parseur calculera automatiquement les voisins par masquage de bits XOR [16] – chaque nœud sera connecté aux nœuds dont l'ID ne diffère que d'un bit (voisins de dimension). Pour une topologie ring, on pourrait avoir un calcul voisin (ex: voisin = id±1 mod N) si non spécifié. En topologie personnalisée, l'utilisateur devra soit lister les voisins pour chaque nœud, soit utiliser les directives @EDGE décrites plus bas. Toute incohérence (par ex. un nombre de voisins incorrect pour la topologie attendue) pourra être signalée.
- `@env <CLE=VALEUR>` : variables d'environnement spécifiques au nœud. **Nouveau** – Cette directive peut apparaître plusieurs fois dans un bloc de nœud pour définir différentes variables d'environnement qui devront être passées au processus/conteneur de ce nœud. Par exemple : `@env MODE=production`. Le parseur stockera ces paires clé=valeur (dans une liste ou un map). Par défaut, un nœud n'a pas de variables d'environnement additionnelles (env vide).
- `@vol <chemin_hôte>:<chemin_conteneur>` : définition d'un volume monté (*volume mount*) pour le nœud. **Nouveau** – Permet de décrire qu'un répertoire ou fichier de l'hôte doit être monté dans le conteneur ou l'environnement isolé du nœud. Par exemple : `@vol /data/input.txt:/app/input.txt` montera le fichier local `/data/input.txt` à l'emplacement `/app/input.txt` dans le conteneur du nœud. Le parseur découpera la chaîne autour des deux points `:` en chemin source et destination, et stockera ces paires dans une liste de volumes (chaque volume pouvant être représenté par une petite structure `Mount{hostPath, containerPath}`). Par défaut, pas de volume monté (liste vide) sauf si précisé.

- `@limits <ressources>` : limites de ressources à appliquer au nœud. **Nouveau** – Spécifie des contraintes comme le CPU ou la mémoire maximale autorisée pour le nœud (utile surtout si on utilise des conteneurs ou cgroups). La syntaxe pourrait être une liste de couples clef=valeur séparés par des espaces ou des virgules. Par exemple : `@limits cpu=2 mem=4G` pour limiter le conteneur à 2 cœurs et 4 Go de mémoire. Le parseur interprétera ces valeurs et les stockera (par ex. dans une structure `Limits{cpu, mem, ...}`). Si non spécifié, aucune limite particulière n'est appliquée (valeurs par défaut nulles ou illimitées).

- **Blocs de job** – Introduisent la description d'un job à exécuter sur le cluster. Un bloc de job commence par la directive `@job` et se termine par `@end`. À l'intérieur, on définit les détails du job, notamment ses phases d'exécution. Les directives prévues sont :

- `@job <Nom>` : ouvre un nouveau bloc décrivant un job. Le nom permet d'identifier le job (il peut servir à l'orchestrateur pour des logs ou à choisir une logique de traitement spécifique). Par exemple : `@job WordCount`. Après cette ligne, toutes les lignes indentées jusqu'à `@end` font partie de ce bloc de job.
- `@input <Chemin>` : spécifie une entrée (fichier ou autre ressource) pour le job. Par exemple : `@input "/data/big.txt"`. Cette information sera stockée (chaîne de chemin, URL, etc.) dans l'objet Job et pourra être utilisée par l'orchestrateur pour fournir les données nécessaires au job. Si le job n'a pas d'input (ou plusieurs), ces directives peuvent être optionnelles ou répétées selon le cas d'usage – ici on suppose un seul input principal par job pour simplifier.
- `@phase <NomPhase> [sélection]` : déclare une phase d'exécution du job, éventuellement associée à un sous-ensemble de nœuds. Chaque phase correspond à une étape du traitement distribuée. *Exemple* : `@phase map [*]` ou `@phase reduce [coordinator]`. Le nom de phase est un identifiant (sans espaces, ou entre guillemets s'il contient des espaces). La **sélection** entre crochets est optionnelle – si elle est présente, elle indique sur quels nœuds la phase doit s'exécuter. Si elle est omise, on pourra considérer par défaut que la phase s'applique à **tous les nœuds** (`[*]`). Le parseur va interpréter la sélection (voir plus bas les raccourcis de sélection) pour stocker la liste des ID de nœuds concernés par cette phase (par ex. phase "map" -> {0,1,2,3,4,5,6,7} et phase "reduce" -> {0} si [coordinator]). Un objet `Phase` contiendra au minimum le nom de phase et la liste des nœuds cibles. On peut également prévoir d'y stocker d'autres paramètres s'ils sont ajoutés plus tard (par ex. code à exécuter, image de conteneur spécifique pour cette phase, etc., mais ces détails ne sont pas explicités dans la question). Dans la configuration actuelle, les phases `map` et `reduce` du WordCount ont tous les nœuds impliqués (chaque nœud fait d'abord un calcul local puis une réduction hypercube) [17] [18] – notre DSL permettra de décrire cela de manière déclarative.

- `@end` : marque la fin du bloc de job. Cette directive de fermeture permet au parseur de savoir que les directives du job se terminent ici. Chaque `@job` ouvert doit avoir un `@end` correspondant. Si plusieurs jobs sont décrits dans le même fichier `.hc`, on aura ainsi plusieurs blocs `@job ... @end` successifs. Une erreur sera levée si un bloc n'est pas refermé correctement.

- **Directives globales supplémentaires** – Ce sont des réglages s'appliquant globalement (en dehors de tout bloc de nœud ou de job). Elles peuvent apparaître n'importe où au niveau supérieur du fichier (généralement avant de définir les nœuds ou les jobs). Parmi elles :

- `@metrics on|off` : activation de la collecte de métriques. Par exemple, `@metrics on` indiquerait à l'orchestrateur de collecter des métriques de performance pendant l'exécution du job (temps de chaque phase, utilisation CPU/mémoire, etc.). Le parseur peut stocker un booléen ou un indicateur de niveau de métrique.
- `@trace on|off` : activation du traçage détaillé. Semblable à metrics, cela pourrait indiquer de tracer finement l'exécution (par ex. pour debugging, enregistrements d'événements). S'il est spécifié sans valeur explicite, on peut supposer que la présence de `@trace` l'active (booléen true).
- `@log <niveau>` : configuration du niveau de log (par exemple `@log debug` ou `@log warn`). Le parseur peut mapper cela à un paramètre interne (niveau d'affichage).
- `@require <condition>` : condition requise pour la configuration. **Nouveau** – Cette directive permet d'exprimer une contrainte qui doit être satisfaite, sinon l'analyse échoue. Par exemple :

`@require @use >= 4` ou `@require @topology == "hypercube"`. Le parseur évaluera la condition (on peut imaginer un mini-évaluateur pour des expressions booléennes sur les paramètres connus, ou imposer des formes simples), et en cas d'échec, il lèvera une erreur explicite. Cela aide à vérifier la validité des fichiers DSL (par exemple exiger un nombre de nœuds minimal, ou la présence d'une directive). **Techniquement**, on peut restreindre la syntaxe à des comparaisons simples entre constantes, variables ou paramètres (==, !=, <, >, etc.). Les @require facilitent l'auto-documentation et la sécurité du DSL.

- `@warn <message>` : avertissement conditionnel ou inconditionnel. **Nouveau** – Sert à afficher un avertissement lors du parsing si une certaine situation se produit, sans interrompre l'exécution. Par exemple : `@warn "Pas de phase de reduce spécifiée, résultat non agrégé."`. Le parseur lorsque il rencontre `@warn` pourrait toujours imprimer le message ou le stocker pour affichage lors du *dry-run*. On peut aussi imaginer une forme conditionnelle (similaire à require, mais n'émet qu'un warning si la condition n'est pas remplie, via une syntaxe du type `@warn if <cond> "<message>"`). Ces warnings ne bloquent pas l'exécution mais informent l'utilisateur.
- `@time on|off` : suivi du temps d'exécution global. Ce réglage pourrait demander à l'orchestrateur de mesurer le temps total du job ou d'insérer des timestamp/logs pour chaque phase. C'est similaire à metrics/trace, on le stockera comme booléen ou niveau de timing.
- `@let <Nom> = <valeur>` : définition de variable. **Nouveau** – Permet d'introduire des variables utilisateur dans le DSL, dont la valeur pourra être injectée plus tard dans le fichier. Par exemple : `@let WORKERS = 7`. Ces variables peuvent représenter des constantes (nombres, chaînes…) pour éviter les répétitions ou calculer des valeurs dérivées. Le parseur stockera ces variables dans une table (dictionnaire) accessible lors du traitement des lignes suivantes. Il faudra gérer leur portée (probablement globales à tout le fichier de config) et l'ordre (une variable doit être définie avant d'être utilisée).

- `@include <fichier>` : inclusion d'un autre fichier DSL. **Nouveau** – Permet de décomposer la configuration en plusieurs fichiers et de les assembler. Par exemple : `@include "cluster_topology.hc"`. Le parseur, en rencontrant cette directive, ouvrira le fichier spécifié et le parsera *récursivement* comme si son contenu était inséré à cet endroit. Ceci facilite la réutilisation de configurations communes (ex: définir un cluster standard dans un fichier, et les jobs dans un autre). On devra veiller à éviter les inclusions circulaires (on peut garder une liste des fichiers déjà inclus pour détecter les boucles et lancer une erreur le cas échéant). L'inclusion peut être gérée en relative par rapport au fichier courant ou via un chemin absolu. Toute erreur dans le fichier inclus sera relayée comme si elle provenait de l'inclusion.

- **Sélecteurs de nœuds et raccourcis** – Pour simplifier la désignation d'ensembles de nœuds (par ex. dans les listes de voisins ou les phases de jobs), le DSL introduit des raccourcis :

- `[*]` : représente **tous les nœuds** du cluster. C'est l'équivalent d'une liste contenant tous les IDs de `0` à `N-1`. Par exemple, `@phase broadcast [*]` signifie que la phase « broadcast » s'exécute sur *tous* les nœuds.
- `[workers]` : représente tous les nœuds dont le rôle est *worker*. Dans un cluster Hamon Cube typique, cela exclut le coordinateur (nœud 0) et ne sélectionne que les nœuds 1,2,… *N-1* (en supposant un seul coordinateur). Ce raccourci permet par exemple `@phase map [workers]` pour indiquer que seuls les workers exécutent la phase « map », tandis que le coordinateur reste inactif sur cette phase (ou joue un rôle différent). Le parseur implémentera cela en filtrant la liste des Node définis et en sélectionnant ceux dont `role` == "worker" ou commence par "custom:" autre que coordinator. *(Nota:* si le DSL supporte plusieurs coordinateurs ou d'autres rôles, on peut généraliser `[workers]` pour *tous sauf coordinateur*, ou éventuellement introduire d'autres sélecteurs par rôle, comme `[coordinator]` pour spécifier uniquement le nœud coordinateur,

ou `[custom:XYZ]` pour tous les nœuds ayant un rôle personnalisé donné. La question ne le mentionne pas explicitement, on se concentre sur `[workers]` tel que demandé.)

- `@DIM(d)` : ce raccourci exploite la notion de dimension du cube. **Deux utilisations** sont envisageables :
    1. *En tant que constante* – `@DIM(d)` peut retourner le nombre total de nœuds correspondant à une dimension donnée *d*. Par exemple, si *d* vaut 3, `@DIM(3)` pourrait être évalué en 2^3 = 8. Cela pourrait être utile en combinaison avec `@use` ou `@require` pour définir la taille du cluster via la dimension. Par exemple : `@use @DIM(3)` serait équivalent à `@use 8`. Si `@dim` a déjà été spécifié, on pourrait aussi utiliser `@DIM` sans argument pour signifier la dimension actuelle. Toutefois, l'intérêt principal est faible car on pourrait directement écrire 8. Plus utilement, on peut traiter `@DIM` comme une variable spéciale contenant la dimension effective du hypercube (après calcul automatique ou fournie). Ainsi `${DIM}` inséré dans une expression donnerait la dimension numérique.
    2. *En tant que sélecteur de nœuds* – On peut imaginer que `@DIM(d)` sélectionne tous les nœuds ayant un certain index binaire spécifique. Cependant, cette interprétation est moins directe. Il est plus probable que `@DIM(d)` soit prévu comme une **macro de calcul** plutôt qu'un sélecteur. Nous l'implémenterons comme un mot-clé qui évalue la valeur de la dimension *d* ou effectue un calcul en fonction de *d*, utilisable dans les expressions ou les conditions. Par exemple, `@require @dim == @DIM(3)` pourrait vérifier qu'on est bien en dimension 3.
- `@EDGE(x->y)` : permet de désigner une connexion entre deux nœuds *x* et *y*. Ce raccourci peut être utilisé de deux façons complémentaires :
    - **Dans la définition de topologie** : en mode topologie libre (`@topology custom`), on peut lister les arêtes du graphe de connexion. Par exemple :

      ```
      @topology custom
      @use 4
      @EDGE(0->1)
      @EDGE(1->3)
      @EDGE(0->2)
      @EDGE(2->3)
      ```

      Ici on décrit un carré (0-1-3-2 connecté en cycle). Le parseur, en rencontrant une directive `@EDGE(a->b)`, ajoutera mutuellement *b* comme voisin de *a* et *a* comme voisin de *b* dans la configuration des nœuds. Cela évite de devoir ouvrir chaque `@node` pour y renseigner les voisins manuellement et assure la cohérence symétrique des voisinages. On vérifie que les IDs donnés existent ou sont dans l'intervalle [0, N-1] défini par `@use`. Si une arête en double est spécifiée, on peut l'ignorer ou émettre un warning. Cette façon de faire permet une flexibilité totale pour définir n'importe quel graphe de connexion.
    - **Dans un contexte de sélection** : on pourrait aussi autoriser `@EDGE(x->y)` à être utilisé là où une liste de nœuds est attendue, par exemple dans un sélecteur de phase ou de voisins. Dans ce cas, on pourrait décider que `@EDGE(x->y)` représente l'ensemble `{x, y}` (les deux extrémités de l'arête). Par exemple, `@phase pair1 @EDGE(2->3)` signifierait une phase impliquant uniquement les nœuds 2 et 3. Cependant, ce raccourci est moins standard, et son utilité est discutable en sélection de phase (on pourrait directement écrire [2,3]). L'usage principal restera pour définir le graphe de topologie. Le terme "raccourci de sélection" suggère néanmoins qu'il peut fournir une liste de nœuds – nous pourrons donc le supporter de cette manière également : lors du parsing d'une liste d'IDs, si un token correspond à la forme `@EDGE(x->y)`, on l'expanse en deux IDs x et y.

En résumé, ces raccourcis de sélection améliorent la lisibilité et réduisent les risques d'erreur. Un utilisateur pourra par exemple écrire simplement `@neighbors [@EDGE(0->1), @EDGE(0->2), @EDGE(0->4)]` plutôt que lister deux fois chaque voisin, ou `@phase compute [workers]` au lieu d'énumérer manuellement tous les workers.

- **Variables et expressions** `${…}` – Le DSL va supporter l'insertion de variables et de calculs simples au sein des valeurs grâce à la notation `${...}`. Cette fonctionnalité apporte de la généricité et évite les répétitions, en permettant par exemple de calculer des ports, des chemins ou des indices en fonction de l'ID du nœud ou de constantes définies :
- **Substitution de variables** : toute chaîne de caractères dans la config contenant une expression `${VAR}` verra `VAR` remplacé par la valeur de la variable correspondante. Par exemple, si on a défini `@let DATA_PATH = "/mnt/data"`, alors dans un bloc de nœud on peut écrire `@vol ${DATA_PATH}/node${id}:/data` pour monter un volume spécifique à chaque nœud. Le parseur détectera `${DATA_PATH}` et remplacera par "/mnt/data", et `${id}` par l'identifiant du nœud courant (ex: "/mnt/data/node0:/data" pour le nœud 0). Les variables à substituer peuvent être :
  - Celles définies par `@let` au préalable (stockées dans une table). On supportera aussi bien des valeurs numériques que des chaînes (il faudra conserver la valeur sous forme de chaîne pour l'insérer telle quelle, ou éventuellement interpréter un nombre si besoin contextuel).
  - Des variables spéciales prédéfinies : on aura notamment `id` (l'ID du nœud courant, utilisable seulement dans les contextes de nœud), possiblement `role` (le rôle du nœud courant), `use` (N, le nombre total de nœuds déclaré), `dim` (la dimension hypercube calculée ou fournie), etc. Par exemple `${use}` serait remplacé par le nombre total de nœuds. Ces variables spéciales permettent d'écrire des choses comme `@require ${use} == 8` ou d'insérer le nombre de nœuds dans un message de warning.
- **Expressions arithmétiques simples** : à l'intérieur de `${...}`, on autorisera des opérations basiques `+` ou `-` avec des constantes ou entre variables. Par exemple `${id+1}` ou `${base_port + id}`. Un cas déjà géré implicitement était l'auto-calcul du port via autoprefix (`HOST:PORT+id`) [11] ; désormais on rend cela explicite et généralisé. Le parseur contiendra un petit évaluateur d'expressions arithmétiques pour ces cas simples : on pourra parse l'intérieur des accolades `${}` en recherchant la présence de `id` ou d'un nom de variable, d'un opérateur +/-, et d'une constante numérique. On supportera des choses comme `${id + 5}` (qui vaudra 5 plus l'ID du nœud), `${NODES - 1}` (si `NODES` est une variable définie, par ex. N-1) ou encore `${dim * 2}` éventuellement si on implémente la multiplication. **Par simplicité**, on peut limiter aux additions et soustractions qui couvrent la plupart des besoins (`*` et `/` pourraient être ajoutés si nécessaire). Ces expressions seront évaluées à un entier ou une chaîne selon le contexte. Par exemple, `${id+1}` dans une string deviendra "1" pour le nœud 0, "2" pour le nœud 1, etc. Nous ferons attention aux priorités et aux espaces (on peut autoriser ou ignorer les espaces autour des opérateurs). En interne, un parseur très simple peut suffire (détecter le motif `<var_or_number> (+|-) <var_or_number>`). Toute variable non définie utilisée dans une expression devra provoquer une erreur.
- **Ordre de définition** : il faudra imposer que `@let` apparaisse avant la première utilisation de la variable correspondante dans le fichier (ou dans un fichier inclus précédemment). Si une variable est référencée sans définition préalable, le parseur lèvera une erreur du type "Variable X non définie". Il pourra être utile de stocker provisoirement les lignes contenant des variables puis de les résoudre en deuxième passe, mais compte tenu de la simplicité, on peut également résoudre à la volée : à chaque ligne lue, avant de la traiter, remplacer toute occurrence de `${...}` par sa valeur calculée (en se basant sur l'état actuel des variables et du contexte). L'approche *single-pass* est viable si les variables sont définies avant usage (ce qui sera documenté comme une

contrainte d'écriture du DSL). Dans les rares cas d'utilisation anticipée, un mécanisme plus sophistiqué pourrait être ajouté, mais ce n'est pas nécessairement requis ici.

En introduisant ces variables et expressions, on gagne en souplesse. Par exemple, l'utilisateur pourrait définir en tête de fichier `@let NB_WORKERS = 7` puis écrire `@use ${NB_WORKERS+1}` pour automatiquement ajuster le nombre total de nœuds (ajoutant le coordinateur). Ou définir un préfixe commun pour des chemins de volumes, etc. Cela rend la configuration plus paramétrable.

Conception de l'implémentation en C++17

Structures de données et organisation (.h/.cpp)

L'implémentation sera structurée en code C++17, sans aucune dépendance externe (utilisation exclusive de la STL) comme c'était le cas du MVP initial [19]. On séparera clairement les déclarations (fichier `.h`) et la logique (fichier `.cpp`) pour améliorer la lisibilité et la maintenabilité à mesure que le DSL s'enrichit.

On définira des classes/structures pour représenter les entités de configuration :

- `Node` (ou `NodeCfg`) – Représente un nœud logique du cluster. Cette structure contiendra les champs existants :
- `int id`,
- `std::string role`,
- `int numa`,
- `int core`,
- `std::string host`,
- `int port`,
- `std::vector<int> neighbors` (IDs des voisins logiques), … ainsi que les nouveaux champs :
- `std::unordered_map<std::string, std::string> env` (ou `vector` de paires) pour les variables d'environnement du nœud,
- `std::vector<Mount> volumes` pour les volumes montés (où `Mount` est une petite struct avec `std::string hostPath; std::string containerPath;`),
- `Limits limits` pour les limites de ressources (où `Limits` peut contenir des champs comme `int cpu_cores; size_t mem_bytes;` etc., ou garder les valeurs brutes sous forme de string si on préfère reporter l'interprétation plus tard).

- `Phase` – Représente une phase d'un job. Champs principaux :

- `std::string name` pour le nom de la phase,
- `std::vector<int> nodes` pour la liste des identifiants de nœuds impliqués dans cette phase. On peut aussi stocker un moyen de représenter la sélection d'origine (par ex. garder un flag si c'était "*" ou "workers"), mais ce n'est pas nécessaire si on a la liste exhaustive. D'autres attributs potentiels, non mentionnés explicitement, pourraient être ajoutés plus tard (par exemple un nom de script ou de conteneur à lancer pour cette phase, un indicateur de parallélisme ou de type de phase, etc.). Pour l'instant, nous nous concentrons sur le nom et la portée en termes de nœuds.

- `Job` – Représente un job complet. Principaux champs :

- `std::string name` pour identifier le job,
- `std::optional<std::string> input` pour le chemin/paramètre d'entrée si présent (ou plusieurs inputs si on décide d'en supporter plusieurs, auquel cas ce serait une collection),
- `std::vector<Phase> phases` pour la liste ordonnée des phases du job. L'ordre de ce vecteur correspondra à l'ordre de déclaration des `@phase` dans le fichier, respectant la séquence d'exécution.

- On peut ajouter d'autres champs globaux au job si nécessaire (ex: un type de job, des paramètres spécifiques, etc.).

- `Config` – Structure englobant la configuration complète parsée depuis un fichier DSL. Elle contient :

- Les paramètres globaux de topologie : `int useN; int dim; std::string topology; std::string hostPrefix; int portBase;` correspondant aux directives `@use`, `@dim`, `@topology`, `@autoprefix`.
- Les flags globaux : booléens ou enums pour `metrics`, `trace`, `logLevel`, `time` etc., initialisés par défaut (éventuellement false ou à un niveau standard) et modifiés si les directives correspondantes sont rencontrées.
- Les **collections d'objets** décrites ci-dessus : `std::vector<Node> nodes; std::vector<Job> jobs;`. Les nœuds seront indexés par leur ID pour accès facile (on peut initialiser le vector de Node de taille N après avoir lu `@use N`, en utilisant des index pour stocker chaque Node dans sa position). On utilisera peut-être un `std::vector<std::optional<Node>>` durant le parse pour remplir au fur et à mesure, puis on convertira en Node finalisé dans un vector simple une fois la taille connue, comme c'était fait initialement [20] [21].
- La table des variables définies par `@let` : par ex. `std::unordered_map<std::string, std::string> variables;`. On stockera les valeurs sous forme de chaîne brute, et on pourra stocker aussi un indicateur de type (entier, string) si on veut affiner. Mais comme on ne fait que substituer textuellement ou évaluer de simples sommes, la chaîne peut suffire (on convertira en int pour les calculs quand nécessaire).
- On peut également avoir une liste de warnings accumulés (`std::vector<std::string> warnings`) pour conserver les messages de `@warn` et éventuellement les réafficher en fin de parsing ou lors du dry-run.

Ces classes seront définies dans le header `HamonDSL.h` (ou `Hamon.h` refactorisé) et les méthodes de parsing/traitement dans `HamonDSL.cpp`. L'API exposée pourrait consister en une fonction ou méthode statique du style `bool parse_file(const std::string& path, Config& config)` qui lit un fichier et remplit la structure Config, en retournant true/false selon succès ou en lançant des exceptions en cas d'erreur. Par exemple, l'utilisation côté orchestrateur serait :

```
HamonDSL::Config cfg;
HamonDSL::parse_file("scenario.hc", cfg);
cfg.finalize();
cfg.print_plan();
```

comme illustré dans le MVP [22], à la différence que `Config` est maintenant enrichi des jobs et autres paramètres. La méthode `finalize()` et `print_plan()` feront partie de `Config` ou d'une classe utilitaire associée.

Logique de parsing (analyse syntaxique)

Le parseur sera implémenté de manière *impérative* en parcourant le fichier ligne par ligne, similaire à l'approche du MVP (basée sur `std::ifstream`, la fonction `getline` et des utilitaires de trim et de test de préfixe) [23] [24]. Étant donné la relative simplicité de la grammaire (un DSL déclaratif, non ambigu, avec une directive par ligne), un analyseur descendant fait main est approprié et évite une dépendance à Flex/Bison ou autre.

Lecture du fichier : On ouvrira le fichier `.hc` et on lira séquentiellement chaque ligne. On tiendra à jour un compteur de lignes (pour d'éventuels messages d'erreur plus précis). Chaque ligne sera d'abord *trimée* (espaces en début/fin supprimés) [23]. Si la ligne est vide ou commence par un caractère de commentaire (par exemple on pourrait décider que `#` ou `//` introduisent un commentaire), alors on l'ignore. Sinon, on identifie la nature de la ligne en regardant son préfixe. Une série de `if/else if` ou une structure `switch` sur des mots-clés peut être utilisée. Le MVP utilisait des helpers comme `starts_with(s, "@use")` [24], on poursuivra dans ce sens.

État de contexte : Puisque notre DSL a des blocs imbriqués (node, job, phase), il est important de savoir dans quel contexte on se trouve pour interpréter correctement la ligne : - On peut avoir un indicateur booléen ou un enum pour le contexte courant, par exemple `enum Context { GLOBAL, NODE, JOB /*, PHASE*/ };` et des variables comme `currentNodeId` et `currentJobIndex` pour savoir sur quel nœud ou quel job on opère. - Lorsqu'on lit `@node X`, on passe en contexte NODE pour ce node X (on stocke `currentNodeId = X`). Toutes les lignes suivantes indentées (dans le fichier, l'indentation est indicative mais notre parseur a trim, donc on ne la voit plus directement) seront traitées tant qu'on ne rencontre pas une nouvelle directive de niveau supérieur (une directive globale, un nouveau `@node`, ou un `@job`). Si on lit un `@job` alors qu'on était dans un node, cela signifierait probablement qu'on a oublié de finir la définition du node précédent. Comme la grammaire ne prévoit pas de `@end` pour un node (les blocs de nœud se terminent implicitement au début d'un nouveau bloc de même niveau), il faudra simplement faire en sorte que toute directive non reconnue comme interne à un nœud ferme le contexte courant. En pratique, on peut gérer ainsi : - Quand un `@node` est lu, on crée/accède au Node correspondant et on marque ce contexte actif. - À chaque nouvelle ligne, si elle commence par `@` et correspond à une directive **valide dans le contexte NODE** (`@role`, `@cpu`, etc.), on la traite pour le `currentNodeId`. Si la ligne commence par `@node` ou `@job` alors que `currentNodeId` n'est pas nul, cela signifie qu'on a terminé la description du nœud précédent. On peut donc d'abord clôturer implicitement le bloc du node courant (par ex. on pourrait mettre `currentNodeId = -1` pour revenir en contexte GLOBAL) et ensuite traiter la nouvelle directive. - Alternativement, on peut simply faire `currentNodeId = -1` chaque fois qu'on termine de remplir un node (par exemple juste après un `@node` block, mais comme on ne sait pas où il se termine explicitement, c'est dès qu'on voit une directive de plus haut niveau). - Pour les blocs `@job`, nous avons une fermeture explicite `@end`. Donc la logique sera : - Quand on lit `@job Name`, on crée un nouvel objet Job (on peut stocker `currentJobIndex` = index de ce job dans `config.jobs` ou stocker un pointeur/ref `currentJob`). On passe en contexte JOB (et on note qu'on n'est plus en contexte Node si c'était le cas, donc on devrait avoir terminé tous les nodes avant d'entamer un job dans la config, ou vice versa – on peut autoriser d'avoir la section jobs après, ce qui semble logique). - À l'intérieur d'un job, on peut encore avoir des sous-contextes (phases), mais comme on a choisi de considérer qu'une phase se définit sur une seule ligne (sans bloc interne propre avec fin séparée), on n'a pas besoin d'un contexte *phase* persistant. Chaque `@phase` lu sera immédiatement ajouté à la liste du job courant. Si dans le futur on voulait permettre un bloc par phase (par ex. pour énumérer des actions), on devrait alors introduire un `@endphase` ou réutiliser `@end` différemment, mais ici ce n'est pas requis. - Quand on lit `@end`, on s'attend à être dans un contexte JOB. La rencontre de `@end` va clôturer le job en cours : on pourra faire `currentJobIndex = -1` ou un flag `inJob = false` pour repasser en contexte

GLOBAL. Toute directive après `@end` sera traitée comme hors bloc (global) ou pourra démarrer un autre bloc (par ex. un autre `@job` ou rien). Si un `@end` survient alors qu'on n'était pas dans un job, ce sera considéré comme une erreur de syntaxe (bloc de job non ouvert). Inversement, si la fin de fichier est atteinte alors qu'un job est toujours ouvert (pas de @end rencontré), on lèvera aussi une erreur de bloc non fermé.

Traitement de chaque directive : En pseudo-code, la boucle principale pourrait ressembler à :

```
int lineNum = 0;
Context ctx = GLOBAL;
int currentNodeId = -1;
Job* currentJob = nullptr;

std::string line;
while (std::getline(file, line)) {
    lineNum++;
    std::string s = trim(line);
    if (s.empty() || s.rfind("//", 0) == 0 || s.rfind("#", 0) == 0)
continue;  // commentaires ou lignes vides

    // Gestion des substitutions variables dans la ligne s avant analyse :
    substituteVariables(s, currentNodeId, currentJob);

    if (starts_with(s, "@use")) {
       // ... extraire l'entier et stocker dans config.useN ...
       config.useN = std::stoi(...);
       continue;
    }
    if (starts_with(s, "@dim")) { ... }
    if (starts_with(s, "@topology")) { ... }
    if (starts_with(s, "@autoprefix")) { ... }
    if (starts_with(s, "@metrics")) { ... }
    // ... idem pour trace, log, etc (set flags in config) ...
    if (starts_with(s, "@let")) {
        // parse 'name = value', store in config.variables map
        continue;
    }
    if (starts_with(s, "@include")) {
        // parse filename, call recursively parse_file on that file
(maintaining current config state)
        // possibly track included files to avoid recursion loops
        continue;
    }
    if (starts_with(s, "@node")) {
        // close current node context if any (though typically nodes would be
defined in one block)
        currentNodeId = parseNodeId(s);
        // ensure Node object exists in config (use ensure_node logic)
        config.ensureNode(currentNodeId);  // similar to existing ensure_node
25
```

```
        ctx = NODE;
        continue;
    }
    if (starts_with(s, "@role")) {
        if (ctx != NODE || currentNodeId < 0) bad("Directive @role hors d'un
bloc @node");
        std::string roleStr = ...; // parse after @role
        config.nodes[currentNodeId].role = roleStr;
        continue;
    }
    if (starts_with(s, "@cpu")) {
        if (ctx != NODE) bad("Directive @cpu hors d'un bloc @node");
        // parse "numa=I core=J"
        int numa, core;
        // ... extraction logic ...
        config.nodes[currentNodeId].numa = numa;
        config.nodes[currentNodeId].core = core;
        continue;
    }
    if (starts_with(s, "@ip")) {
        if (ctx != NODE) bad("@ip hors bloc node");
        std::string addr = ...; // after "@ip "
        parse_host_port(addr, host, port);  // reuse function 14
        config.nodes[currentNodeId].host = host;
        config.nodes[currentNodeId].port = port;
        continue;
    }
    if (starts_with(s, "@neighbors")) {
        if (ctx != NODE) bad("@neighbors hors bloc node");
        std::string listStr = ...; // after "@neighbors "
        std::vector<int> neigh = parse_list_ids(listStr);  // e.g. existing
func 26
        config.nodes[currentNodeId].neighbors = neigh;
        continue;
    }
    if (starts_with(s, "@env")) {
        if (ctx != NODE) bad("@env hors bloc node");
        std::string kv = ...; // e.g. "KEY=VAL"
        auto posEq = kv.find('=');
        if(posEq == std::string::npos) bad("Mauvais format de @env, '='
manquant");
        std::string key = kv.substr(0, posEq);
        std::string val = kv.substr(posEq+1);
        config.nodes[currentNodeId].env[trim(key)] = trim(val);
        continue;
    }
    if (starts_with(s, "@vol")) {
        if (ctx != NODE) bad("@vol hors bloc node");
        std::string pathSpec = ...; // e.g. "/host/path:/cont/path"
        auto posColon = pathSpec.find(':');
        if(posColon == std::string::npos) bad("Format @vol invalide, ':'
```

```
manquant");
        Mount m;
        m.hostPath = pathSpec.substr(0, posColon);
        m.containerPath = pathSpec.substr(posColon+1);
        trim(m.hostPath); trim(m.containerPath);
        config.nodes[currentNodeId].volumes.push_back(m);
        continue;
    }
    if (starts_with(s, "@limits")) {
        if (ctx != NODE) bad("@limits hors bloc node");
        std::string limitsSpec = ...; // e.g. "cpu=2 mem=4G"
        // parse by splitting on space or commas
        Limits lim = {};
        for(const auto& token : split(limitsSpec, ' ')) {
            auto pos = token.find('=');
            if(pos == std::string::npos) continue;
            std::string k = token.substr(0,pos);
            std::string v = token.substr(pos+1);
            if(k == "cpu") lim.cpu_cores = std::stoi(v);
            else if(k == "mem") lim.mem_bytes =
parseMemoryString(v); // e.g. "4G" -> 4*1024*1024*1024
            // etc for other possible limits
        }
        config.nodes[currentNodeId].limits = lim;
        continue;
    }
    if (starts_with(s, "@EDGE(")) {
        // Global edge definition (topology)
        // parse "a->b" inside the parentheses
        int a, b;
        // ... parse logic ...
        config.ensureNode(a);
        config.ensureNode(b);
        // add neighbors symmetrically:
        config.nodes[a].neighbors.push_back(b);
        config.nodes[b].neighbors.push_back(a);
        continue;
    }
    if (starts_with(s, "@job")) {
        // Starting a new job block
        std::string jobName = ...; // parse after "@job"
        config.jobs.emplace_back(Job{.name = jobName});
        currentJob = &config.jobs.back();
        ctx = JOB;
        continue;
    }
    if (starts_with(s, "@input")) {
        if (ctx != JOB || !currentJob) bad("@input hors d'un bloc @job");
        std::string inputVal = ...; // parse after "@input "
        // remove quotes if any
        currentJob->input = trim(removeQuotes(inputVal));
```

```
        continue;
    }
    if (starts_with(s, "@phase")) {
        if (ctx != JOB || !currentJob) bad("@phase hors d'un bloc @job");
        // parse phase name and optional [..] selection
        std::string name, sel;
        // e.g. s = "@phase Map [workers]" or "@phase Reduce"
        // Extract the phase name (until space or end or '[')
        size_t posSpace = s.find(' ');
        if(posSpace == std::string::npos) bad("Nom de phase manquant");
        name = s.substr(posSpace+1);
        trim(name);
        std::vector<int> phaseNodes;
        if (name.find('[') != std::string::npos) {
            // There is a selection
            size_t posBracket = name.find('[');
            sel = name.substr(posBracket); // include '['
            name = trim(name.substr(0, posBracket));
        }
        if (!sel.empty()) {
            phaseNodes = parseSelectionList(sel, config); // custom function
to handle * and workers etc.
        } else {
            // default selection = all nodes
            phaseNodes.resize(config.useN);
            std::iota(phaseNodes.begin(), phaseNodes.end(), 0);
        }
        Phase ph;
        ph.name = name;
        ph.nodes = phaseNodes;
        currentJob->phases.push_back(ph);
        continue;
    }
    if (starts_with(s, "@end")) {
        if (ctx != JOB || !currentJob) bad("Directive @end sans @job");
        // close current job block
        currentJob = nullptr;
        ctx = GLOBAL;
        continue;
    }
    if (starts_with(s, "@require")) {
        // parse condition, evaluate it against current config state
        bool ok = evaluateCondition(conditionStr, config);
        if (!ok) {
            bad("Condition @require échouée : " + conditionStr);
        }
        continue;
    }
    if (starts_with(s, "@warn")) {
        // parse message or condition+message
        std::string warnMsg = ...;
```

```
        // If conditional syntax, evaluate similarly to require but just
collect message if condition true or false accordingly
        config.warnings.push_back(warnMsg);
        continue;
    }
    // ... any other directives ...
    bad("Directive inconnue ou hors-contexte : " + s);
}
```

(Le code ci-dessus est donné à titre illustratif pour montrer la structure ; dans l'implémentation réelle, on gèrera proprement la récupération de sous-chaînes après les mots-clés, peut-être via une fonction d'utilité `get_after(keyword, s)` *etc.)*

Dans cette logique : - On effectue les substitutions de variables **immédiatement** sur la ligne brute `s` avant de tester les mots-clés. La fonction `substituteVariables(s, currentNodeId, currentJob)` chercherait les occurrences de `${...}` dans la chaîne et les remplacerait. Elle utiliserait `config.variables` pour les noms définis via `@let`, et des valeurs spéciales pour `${id}` (en utilisant `currentNodeId` si `ctx == NODE`), `${use}`, `${dim}` (connus après lecture de @use/@dim, donc il faut que ces directives apparaissent avant toute utilisation). S'il y a une expression du type `${id + 1}`, on parse à l'intérieur des accolades pour calculer la somme : par exemple on peut détecter `id` puis `+` puis un nombre. On supportera éventuellement l'ordre `const + id` ou `id + const`. En outre, `substituteVariables` peut gérer plusieurs substitutions dans la même ligne (ex: `"${DATA_PATH}/node${id}"` a deux variables). On veillera à bien gérer les cas d'accolades imbriquées ou caractères spéciaux (peu probable ici). - Chaque directive est analysée selon le contexte : par exemple, si on voit `@role` en dehors d'un bloc de nœud, on génère une erreur. Cela assure qu'on ne met pas de directives au mauvais endroit. - La création d'un nœud via `ensureNode(id)` est semblable à celle du MVP [25] : on agrandit le vecteur de nœuds si nécessaire, on initialise un Node par défaut à l'indice voulu s'il n'existe pas encore. On vérifie aussi que l'ID n'est pas négatif et, si `useN` est connu, qu'il est inférieur à `useN` [27] (sinon erreur du type "Node id out of range"). Si `useN` n'est pas encore défini (ex: si l'utilisateur a listé des nœuds avant @use, on pourrait autoriser mais il faudra alors ajuster useN plus tard – idéalement on exige que @use vienne en premier, ce qui sera recommandé). - Le parsing de la liste de voisins utilise la fonction existante `parse_list_ids` [26], mais il faudra l'enrichir pour reconnaître nos nouveaux raccourcis. Possibilité : écrire une fonction générique `parseSelectionList(string, config)` qui gère : - Enlève les crochets `[...]` en bout de ligne (après trim). - Si le contenu est `"*"` (ou vide après trim), retourne la liste complète de 0 à N-1. - Si c'est `"workers"`, filtre la liste des nodes dont `role` est `"worker"` ou commence par `"custom:"` (et possiblement exclure node 0 s'il est coordinateur). - Si ça correspond à un format de liste de nombres "a,b,c", on délègue à `parse_list_ids` pour extraire les entiers. - Si ça contient la syntaxe d'une arête, par ex `"@EDGE(2->3)"`, on peut soit remplacer ça par "2,3" avant d'appeler parse_list_ids, soit traiter explicitement : détecter `"@EDGE("` puis lire les deux nombres et les ajouter. - Si ça contient `@DIM` ou référence de variable, théoriquement ça devrait déjà avoir été substitué par `substituteVariables` avant, donc ici on devrait plus les voir. - La directive `@include` va appeler récursivement le parseur. Concrètement, on peut implémenter `parse_file(path, config)` comme une fonction qui encapsule la lecture ligne par ligne ci-dessus. En cas de `@include`, on peut appeler une fonction interne `parse_from_stream(stream, config)` qui peut être appelée aussi bien pour le fichier principal que pour chaque fichier inclus. On passera un nouveau ifstream pour le fichier inclus et on appellera `parse_from_stream(f_included, config)` depuis l'intérieur de la loop. Il faudra sauvegarder l'état de `currentNodeId`, `currentJob` et du contexte avant d'entrer dans l'inclusion, au cas où l'inclusion est faite à un endroit particulier (par exemple en plein milieu d'un job ou d'un node, bien que cela ne soit pas conseillé, on peut l'autoriser). On peut restreindre en spécifiant que

`@include` n'est autorisé qu'au niveau global pour éviter des complications. - Les `@require` et `@warn` qui contiennent des expressions conditionnelles seront parsés avec un petit évaluateur. On peut permettre un sous-ensemble de syntaxe, par ex. `@require X == Y` ou `@require NODES >= 8`. Le parseur peut utiliser les variables et paramètres connus : on remplacera `NODES` par `useN`, etc., puis on évalue la condition booléenne. Si c'est faux, on utilise `bad()` pour les require (arrêt avec exception) ou on stocke un warning pour les warn.

Gestion des erreurs et robustesse

Un point crucial est de rendre le parseur **robuste**, c'est-à-dire capable de détecter proprement les erreurs de syntaxe ou de structure et d'en informer clairement l'utilisateur. Pour cela : - On améliorera la fonction d'erreur `bad(msg)` déjà utilisée [28] pour qu'elle indique idéalement le numéro de ligne et le contenu fautif. Par exemple, au lieu de seulement `throw std::runtime_error(msg)`, on peut faire :

```
throw std::runtime_error("Erreur ligne " + std::to_string(lineNum) + ": " +
msg);
```

Ceci facilitera le debugging des fichiers DSL mal formés. - **Erreurs de contexte** : comme illustré plus haut, si on trouve une directive dans un endroit inapproprié, on lève une erreur. Exemples : - `@role` sans qu'un `@node` n'ait été ouvert auparavant. - `@phase` en dehors d'un bloc `@job`. - `@end` sans bloc correspondant. - Un second `@job` est démarré alors qu'un précédent n'est pas encore fermé par `@end`. - **Erreurs lexicales** : mauvais format d'une directive : - Valeur manquante (ex: `@use` sans nombre, `@cpu` sans `numa=...`). - Caractères inattendus (ex: `@neighbors [1,2,,3]` avec une virgule en trop). - Nom de variable non valide ou non défini dans `${}`. - Fichier d'include introuvable (chemin incorrect) – on lèvera une erreur "Impossible d'inclure le fichier X". - Dans `@limits`, un type de ressource inconnu (ex: `io=...` non supporté – on peut soit ignorer soit prévenir). - Toute directive inconnue (typo) sera captée par le dernier cas `bad("Directive inconnue ...")`. - **Erreurs de logique** : on profite du parse pour vérifier la cohérence : - Si après lecture complète, `useN` n'a pas été renseigné ou est <= 0, on lance une erreur "Missing or invalid @use N" (message déjà utilisé) [29]. - Si la topologie est hypercube mais `useN` n'est pas une puissance de deux et que `dim` n'a pas été explicitement donné, on lance une erreur conforme [7] (« *@topology hypercube requires @use to be power of two or set @dim explicitly* »). De même, on peut vérifier qu'un `@dim` donné correspond bien à `useN` (c.-à-d. que 2^dim == useN) sinon lever un avertissement ou recalcule auto. - Si la topologie est ring et qu'un certain check s'applique (peut-être aucune contrainte stricte sur N, mais par exemple on peut s'attendre à ce que chaque nœud ait 2 voisins), on peut valider cela. - Vérifier que chaque nœud a bien un rôle déterminé (sinon appliquer défaut coordinateur/worker déjà fait dans finalize). - Vérifier qu'aucun nœud requis n'est manquant : par exemple si `@use 8` mais que l'utilisateur n'a défini explicitement que les nodes 0 à 6 et pas 7, on crée le 7 par défaut dans finalize [30]. On peut ajouter un warning dans ce cas pour signaler "Le nœud 7 n'était pas défini, il a été créé avec les valeurs par défaut". Ainsi l'utilisateur sait qu'il a peut-être oublié de configurer quelque chose pour ce nœud. - Concernant les jobs, on peut vérifier : - Qu'au moins une phase est définie dans un job (sinon job vide sans effet). - Que les phases ont des noms uniques au sein d'un job (deux phases du même nom dans un job pourraient prêter à confusion – sauf si c'est autorisé pour répéter un même type de phase, mais on peut supposer unique par job). - Si une phase utilise un sélecteur de nœuds vide (par ex. `[workers]` alors qu'aucun nœud n'est worker, ou `[roleX]` qui ne correspond à rien), on peut lever un warning ou une erreur. Par exemple, si node0 est coordinateur et nodes1-7 sont workers, `[workers]` va sélectionner {1,...,7} qui est bon, mais `[coordinator]` donnerait {0}. On peut supporter `[coordinator]` comme synonyme de [0] pour être complet. - Si un `@input` est fourni

pour un job, s'assurer que le chemin n'est pas vide. Si plusieurs `@input` dans le même job, on peut concaténer ou garder le dernier, mais mieux vaudrait traiter plusieurs inputs en liste (non spécifié, on considère un seul principal). - Gérer les conflits ou doublons : Si une variable `@let` est définie deux fois avec des valeurs différentes, on peut soit autoriser la redéfinition (en mettant à jour la valeur) soit émettre un warning. Il est sans doute plus propre de considérer cela comme une erreur (une variable ne devrait pas être redéfinie, sauf si on autorise des local overrides dans des includes, mais restons simples). - Si `@include` est utilisé de façon récursive (fichier A inclut B qui inclut A), il faut détecter la boucle. On peut conserver un ensemble de fichiers visités dans le `Config` ou en static du parseur pour refuser d'inclure un fichier déjà en cours d'inclusion.

L'idée est de **détecter le plus tôt possible** les erreurs et d'arrêter le parsing en lançant une exception (ou en retournant false après avoir loggé l'erreur). On utilisera les exceptions (std::runtime_error) comme dans le MVP [7] [20], car cela simplifie la gestion en cas d'erreur fatale. Chaque appel de `bad(msg)` lèvera, pouvant être attrapé plus haut si on veut afficher proprement.

Finalisation des valeurs par défaut

Une fois le fichier parcouru sans erreur syntaxique, on appelle `config.finalize()` pour compléter les valeurs manquantes et effectuer des validations finales, comme c'était fait initialement [31]. Voici les principales actions de finalisation, combinant l'existant et les nouveautés : - **Compléter les nœuds manquants** : si `@use N` était défini et que certains nœuds de 0 à N-1 n'ont jamais été mentionnés par `@node`, on les a déjà créés via `ensureNode` lors du parse, mais `ensureNode` ne pouvait être appelé que quand on rencontrait un id explicitement. Il faut donc s'assurer que la taille du vecteur `nodes` est bien N et que chaque entrée est initialisée [21]. Le MVP faisait :

```
if (config.size() < useN) config.resize(useN);
for i in 0..useN-1:
  if !config[i].has_value(): config[i] = NodeCfg(id=i);
```

Nous reproduirons cela [21], adapté à notre nouvelle structure. Tous les nœuds inexistants seront créés avec leurs valeurs par défaut (rôle vide, etc. qui seront fixés juste après). - **Calcul de la dimension** : si la topologie est hypercube et `dim` n'était pas spécifié, on l'a calculé si possible. On double-vérifie ici que `useN` est cohérent avec `dim`. Si `dim` est fourni, on pourrait recalcule `expectedN = 2^dim` et, si `useN` diffère, soit on ajuste useN, soit on lance une erreur. Mieux vaut lancer une erreur si incohérent (ex: l'utilisateur met `@use 6 @dim 3` qui conflict, on signale). Dans le cas hypercube sans `dim`, on a déjà vérifié la puissance de deux et calculé dim = log2(useN) [7]. Pour une topologie ring ou custom, `dim` n'a pas vraiment de sens – on peut ignorer dim s'il est présent ou émettre un warning du genre "Ignoring @dim for non-hypercube topology". - **Appliquer autoprefix** : parcourir chaque Node : - Si un nœud n'a pas de host (adresse) définie, on lui affecte le host du autoprefix s'il y en a un, sinon `"127.0.0.1"` par défaut [9]. De même pour le port : si autoprefix est fourni (hostPrefix non vide et portBase >=0) on fait `node.port = portBase + node.id`, sinon port = 8000 + id [32]. Si un host avait été spécifié mais pas de port, on utilise autoprefix portBase+id ou défaut 8000+id [33]. - Grâce à cela, l'utilisateur peut simplement écrire `@autoprefix 192.168.0.100:9000` et éviter de mettre @ip sur chaque nœud, tout en pouvant surcharger au besoin un cas particulier (celui-là gardera son host/port custom). - **Appliquer les rôles par défaut** : pour chaque Node dont `role` est vide, on assigne `"worker"` sauf si son id est 0 où on assigne `"coordinator"` [12]. Cela reproduit la logique établie. - **Appliquer CPU/NUMA par défaut** : si `numa` == -1, on met 0; si `core` == -1, on peut soit mettre node.id (comme MVP) [12], soit ne rien faire de particulier. Dans l'exemple YAML, ils ont mis les 4 premiers nœuds sur NUMA0 et les 4 suivants sur NUMA1 [34] [35]. Notre simple défaut (core = id, numa

= 0 toujours) n'imite pas exactement cela, mais c'est mentionné que c'est un *hook* pour plus tard [36]. On peut améliorer légèrement le défaut : par exemple, si useN >1 et topology hypercube, on sait que l'hypercube se coupe en deux moitiés sur chaque dimension supplémentaire. On pourrait dire que si id >= useN/2, numa=1 sinon 0 (ce qui correspond à l'exemple où 0-3 sur socket0, 4-7 sur socket1) – mais ce serait figer l'hypothèse 2 sockets. Comme c'est très spécifique hardware, on peut garder la règle simple (tout sur NUMA0) et laisser à l'utilisateur la liberté de mettre explicitement `@cpu numa=...` pour ajuster. On documentera ceci si besoin. - **Vérification finale des voisins** : après avoir rempli tous les nœuds, on peut repasser sur chaque `Node` et si son vecteur `neighbors` est vide, calculer automatiquement en fonction de `topology` : - Pour `hypercube`, utiliser l'algorithme par XOR mask sur l'id [37] (c'est ce que faisait `initializeTopology()` dans l'ancien code, on peut intégrer cette logique ici). On itère d=0 à dim-1 et ajoute `i XOR (1<<d)` comme voisin [37]. Bien entendu, il faut que `useN == 2^dim`, ce qu'on a assuré. Si l'utilisateur avait défini certains neighbors manuellement et laissé d'autres vides, on peut soit ne calculer que pour ceux restés vides, soit recalculer tous et *écraser* ceux définis? Non, il ne faut pas écraser ce que l'utilisateur a précisé. Donc : si un nœud a déjà des neighbors définis (via DSL ou via @EDGE), on le laisse tel quel (considérant que l'utilisateur sait ce qu'il fait). S'il n'en a pas et topologie connue, on auto-remplit. Pour topologie `ring`, si neighbors vide, on ajoute `(id-1 mod N)` et `(id+1 mod N)`. - Si topologie `custom` et qu'aucun neighbors n'est donné pour un nœud, on le laisse vide (il sera isolé) ou on peut considérer ça comme potentiellement une erreur si la config exige connectivité – mais on ne l'imposera pas ici. - **Nettoyage/tris éventuels** : s'assurer que les listes de neighbors n'ont pas de doublons, trier les IDs par ordre croissant éventuellement pour cohérence. De même pour les phases, on peut trier la liste des nœuds d'une phase ou la laisser dans l'ordre d'écriture (pas très important). - **Validation finale des phases** : on peut ici vérifier qu'au moins un job est défini (sinon peut-être alerter "Aucun job n'a été défini dans le fichier" – ce n'est pas forcément une erreur fatale car on pourrait vouloir juste configurer les nœuds). On peut aussi vérifier les conditions d'équilibrage: par exemple, s'il n'y a qu'un coordinateur et qu'une phase est indiquée sur [coordinator] uniquement, avertir que c'est purement séquentiel. Ce genre de vérification relève plus du conseil que de l'erreur.

Toute cette finalisation se fait dans `Config::finalize()`. Elle utilise des informations de la passe de parsing stockées dans `Config`. Par exemple, l'autoprefix aura mis `hostPrefix` et `portBase`, les nodes seront en partie remplis.

Rendu du plan final (mode *dry-run*)

Une exigence est de permettre un *dry-run* du parseur qui imprime le plan final (topologie et déroulement) sans lancer l'exécution. Cela sert à valider visuellement que la configuration interprétée correspond bien aux intentions. Le MVP fournissait déjà une méthode `print_plan()` pour afficher la topologie résolue [38]. Nous allons l'étendre pour qu'elle inclue également les informations sur les jobs.

Le format d'affichage pourrait être, par exemple :

```
Cluster: 8 nodes, topology=hypercube (dim=3)
Nodes:
 - Node 0: role=coordinator, cpu_core=0, numa_node=0,
endpoint=127.0.0.1:8000, neighbors=[1,2,4]
 - Node 1: role=worker, cpu_core=1, numa_node=0, endpoint=127.0.0.1:8001,
neighbors=[0,3,5]
 ... (tous les nœuds de 0 à 7)
Jobs:
 - Job "WordCount":
```

```
    Input: "/data/big.txt"
    Phases:
      * map -> nodes [0,1,2,3,4,5,6,7]
      * reduce -> nodes [0,1,2,3,4,5,6,7] (hypercube all-reduce, final result
on node 0)
 - Job "Backup":
    Input: "/output/result.bin"
    Phases:
      * send -> nodes [workers]  (nodes [1,2,3,4,5,6,7])
      * save -> nodes [coordinator]  (node [0])
```

(Ceci est un exemple inventé illustrant deux jobs, dont un WordCount style MapReduce et un autre fictif).

L'impression sera formattée avec des indentations et puces pour être lisible. On pourra utiliser `std::cout` ou mieux `std::ostringstream` pour composer la sortie (selon qu'on veut la retourner sous forme de string ou l'afficher directement).

Pour chaque nœud, on affiche ses paramètres principaux : ID, rôle, CPU core, NUMA, IP:port, et éventuellement les voisins. S'il a des env, volumes ou limits non vides, on peut les afficher aussi, par exemple :

```
    env: { MODE=production, RETRIES=3 }
    volumes: { "/data/node0":"/app/data" }
    limits: { cpu=2, mem=4096MB }
```

pour signaler les spécificités. On peut décider de n'afficher ces détails que s'ils sont présents pour ne pas surcharger la vue. Le YAML fourni montrait justement toutes ces infos pour chaque nœud de manière assez verbeuse [2] [39]. Avec notre DSL print, on souhaite quelque chose de similaire en lisibilité.

Pour chaque job, on affiche son nom, son input (si présent) et chaque phase avec la liste des nœuds cibles. On peut résoudre les raccourcis dans l'affichage : par ex montrer entre parenthèses quels nœuds ça représente pour `[*]` ou `[workers]`. L'exemple ci-dessus le fait. S'il y a des warnings collectés (via `@warn`), le mode dry-run peut les afficher à la fin, par exemple :

```
Warnings:
 - Pas de phase @end de job de nettoyage prévue (simple remarque)
```

ou afficher sur stdout au fur et à mesure.

En offrant ce retour, on permet à l'utilisateur de vérifier par exemple que le coordinateur a bien été affecté au core 0 NUMA0, que tel volume est bien pris en compte sur le bon nœud, que la phase "reduce" n'embarque pas par erreur le coordinateur si ce n'était pas voulu, etc.

Intégration dans l'orchestrateur Hamon Cube

Une fois le parseur étendu et le DSL enrichi, l'orchestrateur pourra utiliser la structure `Config` résultante de manière similaire qu'avec le YAML initial, mais avec plus de capacités. D'après la

documentation, l'orchestrateur lit la config (YAML ou DSL) afin de *lancer automatiquement chaque instance avec la bonne configuration* [40] . Concrètement, notre `Config` fournit :

- La liste des nœuds avec pour chacun son id, son core/numa d'affectation, et la liste de ses voisins et endpoints pour établir les communications [41] . L'orchestrateur créera pour chaque entrée de `config.nodes` un processus ou conteneur (par exemple en instanciant un objet `HamonNode` ou en lançant un binaire) avec ces paramètres. Grâce aux champs `env`, `volumes`, `limits`, il pourra, si on utilise des conteneurs Docker par exemple, passer ces options (variables d'environnement, montages de volumes, contraintes CPU/mémoire) lors du lancement du conteneur. Dans le cas de simples processus, il pourra set des variables d'environnement via `exec` ou configurer des cgroups pour les limites.
- Le plan d'exécution des jobs avec leurs phases. L'orchestrateur pourra ainsi automatiser la *séquence* du job :
- Par exemple, pour un job de type MapReduce (WordCount), il va lire `config.jobs[0]` supposons. Il voit une phase "map" sur [*] *: il enverra à tous les nœuds un ordre de commencer la phase map (par ex. en envoyant une commande réseau ou en créant un thread dans chaque processus – selon l'implémentation de HamonNode). Dans le log d'exécution fourni, le coordinateur lit l'input et distribue des tâches aux workers qui font leur part* [17] *, c'est cette phase map. Ensuite il voit une phase "reduce" sur [*] également : il coordonne la phase de réduction hypercube en signalant aux nœuds de procéder aux échanges/réductions [42] [43] . Comme le DSL le décrit, l'orchestrateur n'a plus besoin d'être câblé en dur pour quelles phases existent – on peut imaginer des conditions : s'il voit une phase nommée "reduce", il peut appeler une routine générique de réduction en hypercube. S'il voit une phase nommée autrement, possiblement il faut qu'il sache quoi faire (soit via le nom, soit via un mapping externe). Dans le cas d'un job Kpack de build d'image, les phases pourraient être "fetch", "build", "push" par ex., et l'orchestrateur aurait des procédures correspondantes.
- Si un job a une phase réservée aux `[workers]` seulement, l'orchestrateur enverra l'instruction qu'aux nodes workers, tandis que le coordinateur peut rester en attente ou effectuer autre chose. Par exemple, on aurait pu décider que le coordinateur ne fait pas de map, juste supervise, alors DSL : `@phase map [workers]`. L'orchestrateur alors n'envoie pas de chunk de donnée au coordinateur, conformément à la config.
- De même, pour une phase marquée sur `[coordinator]` uniquement, l'orchestrateur saurait qu'il ne doit exécuter cette étape que sur le node 0, les autres restent inactifs ou en attente. Ceci pourrait servir pour une phase finale d'agrégation ou de sauvegarde du résultat (ex: phase "save" où seul le coordinateur écrit le résultat final sur le stockage externe).
- Les réglages globaux comme `metrics` ou `trace` permettront à l'orchestrateur de décider d'activer certaines fonctionnalités transversales :
- Par exemple, si `metrics` est true, il pourrait instancier un collecteur de métriques ou demander à chaque node d'envoyer ses stats à la fin.
- Si `log = debug`, il pourrait augmenter le verbosité des logs dans la console (afficher plus de détails).
- Si `time` est on, il pourrait mesurer le temps global du job ou de chaque phase et l'imprimer.
- Les directives `@require` et `@warn` n'affectent pas directement l'orchestrateur, elles sont plutôt destinées à la validation du fichier. Cependant, on peut imaginer que certains `@warn` informent l'orchestrateur de conditions potentiellement suboptimales (mais le plus souvent un warning est juste loggé lors du parse). Par exemple, un `@warn "Using all nodes for reduce may cause high sync overhead"` pourrait apparaître dans le plan dry-run pour informer l'utilisateur, sans que l'orchestrateur ait à en tenir compte.

En intégrant ce parseur dans le projet Hamon Cube, on améliore la **déclarativité** et la **reproductibilité** de la configuration. Au lieu d'éditer du code ou un YAML à la main, l'utilisateur peut écrire un scénario

dans le DSL, le vérifier avec `print_plan()` et ensuite l'exécuter avec l'orchestrateur en toute confiance. On rejoint ainsi l'approche déclarative comparable à Kubernetes Manifests ou au YAML initial, tout en ayant un langage dédié plus concis et extensible [44]. L'**orchestrateur** n'a plus qu'à consommer l'API de `Config` : par exemple, lancer une boucle pour `for (auto& node : config.nodes) { launch_process(node.id, node.host, node.port, node.role, node.numa, node.core, node.env, node.volumes, node.limits); }` puis orchestrer les `config.jobs` séquentiellement (ou potentiellement en parallèle si on supporte plusieurs jobs à la fois).

Ce nouveau parseur C++ respectera les contraintes de performance et d'indépendance (fichier source et header propres, utilisables sur n'importe quelle plateforme avec la STL). Il permettra d'ajouter facilement de nouvelles directives si besoin car la structure est modulaire. Ainsi, la configuration **HIC/HID** (Hypercube Interconnect/Implementation) est centralisée dans un fichier DSL lisible, adaptable (par exemple on pourra ajouter *la quantité de mémoire allouée, des chemins de volumes* pour chaque nœud [45] comme on l'a fait, ou d'autres paramètres spécifiques au job), garantissant une orchestration cohérente et reproductible sur un ou plusieurs hôtes.

Sources: Les caractéristiques de la version MVP du parseur Hamon DSL et les besoins d'extension sont issus du code et documents du projet Hamon Cube [1] [39]. Le plan maintient les comportements par défaut établis (assignation auto des voisins en hypercube, ports incrémentés, rôle par défaut, etc.) [7] [12] tout en incorporant les suggestions d'amélioration (paramètres de ressources, volumes, etc.) évoquées dans la documentation [45]. L'approche globale vise à offrir un langage puissant mais sûr, contrôlé par l'orchestrateur local, comme décrit dans le contexte du projet [40]. En somme, ce nouveau parseur fournira une base solide pour configurer et lancer des expérimentations distribuées complexes (telles que le job Kpack sur 8 nœuds logiques) de façon déclarative et modulable. [39] [41]

---

[1] [4] [5] [6] [8] [10] [11] [13] [16] [19] [22] [31] [36] [38] Hamon.h
file://file-N3WEL3o3phRfrJH5C8UqpU

[2] [3] [34] [35] [39] [40] [41] [44] [45] Exécution distribuée d'un job Kpack sur 8 nœuds logiques (architecture « Hamon Cube »).pdf
file://file-V7CnLz8BKVf4Evjo2jEr2j

[7] [9] [12] [14] [15] [20] [21] [23] [24] [25] [26] [27] [28] [29] [30] [32] [33] Hamon.cpp
file://file-EhLWYGB6sGitFWY3onLWva

[17] [18] [42] [43] hamon.txt
file://file-CTTLWAE1PEHfkS4ady7aFh

[37] HamonCube.cpp
file://file-4LsPprdjTPiDYUYThSaFaz