# Answer Set Programming for HPC Dependency Solving

Κωνσταντίνος Καϊμάκης (mtn2508)
Γιώργος Νάζος (mtn2519) Γιώργος Πλέσσιας (mtn2524) Ορέστης Τσαγκέτας (mtn2527)

12 Φεβρουαρίου 2026

# Δομή

- Γιατί το HPC κάνει το πρόβλημα δύσκολο
- Spack specs και concretization
- Μοντελοποίηση σε ASP: facts / rules / constraints
- Virtual dependencies, conditional dependencies, variants
- Βελτιστοποίηση + reuse
- Αποτελέσματα / περιορισμοί

# HPC dependency resolution

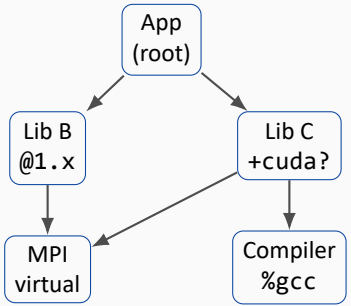
## HPC Stack

- Εκδόσεις: πολλαπλές εκδόσεις συνυπάρχουν
- Compilers: GCC $\Rightarrow$ ABI ασυμβατότητες
- Variants: `+mpi`, `+cuda`, `~debug`, …
- Targets: Skylake/Power9/GPU, …

## Το πρόβλημα

Ο γράφος είναι DAG, αλλά κάθε κόμβος είναι package και τα edges είναι dependencies. Το γενικό πρόβλημα είναι NP-complete.

## Διαισθητικό διάγραμμα




Κάθε βέλος = περιορισμοί.
Κάθε κόμβος = επιλογές/παραλλαγές.

# Υπάρχοντες HPC package managers

## Single prefix package managers

- κοινό prefix $\Rightarrow$ 1 έκδοση/πακέτο
- πιο «στενός» χώρος αναζήτησης από HPC

## Language managers

- συχνά αγνοούν system deps
- ad-hoc επίλυση

## Το παλιό Spack

- αποφάσεις «χωρίς επιστροφή»
- false negatives: αποτυγχάνει ενώ υπάρχει λύση

## Τι θέλουμε

- Πληρότητα: αν υπάρχει λύση, να βρεθεί
- Βελτιστότητα: εύρεση βέλτιστης λύσης
- Συντηρησιμότητα: κανόνες, όχι complex heuristics

# Spack spec syntax

## Περιορισμοί

- @ έκδοση: `hdf5@1.10.2`
- % μεταγλωττιστής: `%gcc@9.3.0`
- +/~ variants: `+mpi`, `~shared`
- ^ εξάρτηση: `^mpich@3.3`

## Παράδειγμα

```
spack install hdf5@1.10.2+mpi %gcc@9.3.0 ^mpich@3.3
```

## Concretization

Η διαδικασία μετατροπής ενός Abstract Spec σε Concrete Spec έτσι ώστε: (i) να ικανοποιούνται οι περιορισμοί και να βελτιστοποιούνται τα κριτήρια, (ii) να επιλύονται όλα τα virtual dependencies, και (iii) να μην υπάρχουν συγκρούσεις.

# ASP Implementation

## Declarative μοντέλο

- Facts: δεδομένα
- Rules: παραγωγή γνώσης
- Constraints: εγκυρότητα

## Ροή

- Python $\rightarrow$ παράγει facts μέσω του Package DSL
- Grounding $\rightarrow$ προτασιακό πρόγραμμα
- clingo $\rightarrow$ stable models

## Παράδειγμα pipeline


## Γιατί όχι SAT;

Η ASP εγγυάται τερματισμό και επιτρέπει τη μοντελοποίηση multi-objective optimization με φυσικό τρόπο.

# Παράδειγμα

## Πακέτα, εκδόσεις, deps

Η Python μεταφράζει τη γνώση των `package.py` σε facts, π.χ.:

```
package(”zlib”).
version_declared(”zlib”,”1.2.11”,0).
possible_dependency(”hdf5”,”mpi”).
```

## Προέλευση των Γεγονότων

- Metadata
- Απαιτήσεις χρήστη
- Κατάσταση συστήματος

## Σχόλιο

Για ένα ρεαλιστικό πρόβλημα δημιουργούνται δεκάδες χιλιάδες facts πριν καν ξεκινήσει το solving.

# Κανόνας επιλογής κόμβων

## Choice rule

```
1{version(P,V):possible_version(P,V)}1:-node(P).
```

## Τι σημαίνει

- για κάθε node(P) επιλέγουμε ακριβώς 1 V
- αποκλείει «διπλές» εκδόσεις στον ίδιο κόμβο λύσης

## Σύγκριση

Αντί για διαδικαστικό backtracking, το backtracking γίνεται εσωτερικά στον solver.

# Κατασκευή γράφου

## Propagation rule

```
node(Dep):-node(Pkg),depends_on(Pkg,Dep).
```

## Κέρδος

Ο γράφος «χτίζεται» λογικά και ο solver εξερευνά ταυτόχρονα τις επιλογές σε βάθος, χωρίς να εγκλωβίζεται από πρώιμες επιλογές.

## Στην πράξη

Αυτό είναι η βάση για την πληρότητα: αν υπάρχει λύση, θα βρεθεί.

# Virtual dependencies

## Ιδέα

Για δυνατότητες τύπου MPI, δεν ζητάμε «πακέτο», ζητάμε λειτουργία. Ο solver επιλέγει provider.

## Κανόνας

```
1{provider(V,P):provides(P,V)}1:-node(P)
,depends_on(P,V).
```

## Παράδειγμα

- hdf5 depends on mpi
- providers: mpich, openmpi
- επιλογή βάσει προτιμήσεων + συγκρούσεων

## Σημείο-κλειδί

Η επιλογή provider είναι μέρος της ίδιας ενιαίας βελτιστοποίησης.

# Εξάρτηση από variant

## Πρόβλημα

Ο παλιός αλγόριθμος αποφάσιζε τις τιμές των variants πριν εξετάσει τις εξαρτήσεις. Πιθανό να οδηγησεις σε αδιέξοδο.

## ASP

Ο solver αποφασίζει ταυτόχρονα τις τιμές των variants και το αν ενεργοποιείται η αντίστοιχη εξάρτηση.

```
condition_holds(ID) :- condition(ID), node(P),
variant_value(P, bzip2, true),
condition_requirement(ID, ”variant_value”, P,
”bzip2”, ”true”).

dependency_enabled(P, bzip2) :-
condition_holds(ID),
dependency_condition(ID, P, bzip2).
```

## Γιατί είναι δυνατό

Ο solver εξετάζει όλο τον χώρο αναζήτησης και κάνει backtracking όπου χρειαστεί.

## Αποτέλεσμα

- λιγότερα αδιέξοδα
- πιο «σωστές» λύσεις σε σύνθετα specs

# Λεξικογραφική βελτιστοποίηση

## Ιδέα

Δεν αρκεί «να υπάρχει λύση». Θέλουμε λύση που να ακολουθεί ιεραρχημένες προτιμήσεις.

## Ενδεικτικά κριτήρια

- Deprecated versions: αποφυγή μη ασφαλών/παρωχημένων
- Version age (roots): νεότερες εκδόσεις για roots
- Variant defaults (roots): τήρηση defaults όπου γίνεται
- Preferred providers: π.χ. MPICH αντί OpenMPI

## Συνέπεια

Κριτήριο 1 υπερισχύει απόλυτα του 2: κανένα trade-off που «σπάει» την ασφάλεια.

## Συνοχή στο HPC

- Compiler mismatch $\downarrow$ (ABI)
- Target mismatch $\downarrow$

# Κριτήρια Βελτιστοποίησης

Ο solver ελαχιστοποιεί τα παρακάτω κριτήρια με λεξικογραφική σειρά:

## Ασφάλεια (1)

Χρήση deprecated εκδόσεων.

## Root Πακέτα (2–5)

- Παλαιότητα εκδόσεων, μη-προτιμώμενοι πάροχοι.
- Αποκλίσεις από τις προεπιλεγμένες τιμές των variants.

## Εξαρτήσεις / Non-roots (6–7, 11–12)

- Παλαιότητα εκδόσεων και variant/provider defaults για τα πακέτα του υπόλοιπου γράφου.

## Συνοχή Συστήματος (8–10, 13–15)

- Mismatches: Ασυμφωνίες σε compiler, OS και target μεταξύ εξαρτήσεων.
- Preferences: Επιλογή μη-προτιμώμενων compilers, OS ή targets.

## Σημείωση

Οι προτιμήσεις των root πακέτων υπερισχύουν πάντα των εξαρτήσεων («flow downward»).

# Resuability & Buildcache

## Το «κόλπο» με τους κάδους

Διπλασιάζουμε κριτήρια: άλλα για νέα builds και άλλα για εγκατεστημένα πακέτα, και εισάγουμε ενδιάμεσα στόχο: ελαχιστοποίηση builds.

## Γιατί έχει σημασία

- builds στο HPC είναι ακριβά
- reuse = γρηγορότερα installs ⇒ καλύτερο UX

## Λογική πολιτική

Αν ένα ήδη εγκατεστημένο ή προ-μεταγλωττισμένο πακέτο ικανοποιεί τους περιορισμούς, ο solver το επιλέγει για να εξοικονομήσει χρόνο και πόρους.

# Εφαρμογή σε E4S και HPC συστήματα

## Περιβάλλον

- E4S repository
- Quartz και Lassen

## Κλιμάκωση με reuse

Ακόμη και με $\sim$63.099 εγκατεστημένα πακέτα, ο solver παραμένει γρήγορος.

## Κύρια ευρήματα

- Solve times: συνήθως $< 1 - 10$ sec
- χρόνος $\uparrow$ με πολλές εναλλακτικές εξαρτήσεις
- ρύθμιση clingo tweety καλύτερη

## Bottleneck;

Μετατοπίζεται στην Python setup phase, όχι στο ίδιο το solving.

# Αποτελέσματα

## Τι άλλαξε ουσιαστικά

- επιλύει σενάρια όπου ο greedy αποτυγχάνει
- καλύτερος χειρισμός: συγκρούσεις εκδόσεων + conditionals
- λύσεις εγγυημένα βέλτιστες βάσει κριτηρίων

## Συντηρησιμότητα

Κανόνες ASP $\approx$ «γνώση» του συστήματος. Πιο καθαρό από heuristics που ξεφεύγουν με τον χρόνο.

## Σύνοψη σύγκρισης

| Χαρακτηριστικό | ASP concretizer |
|---|---|
| Πληρότητα | Ναι |
| Βελτιστοποίηση | Πολυκριτηριακή |
| Variants/Conditionals | Φυσικά ενσωματωμένα |
| Συντηρησιμότητα | Υψηλή |

# Περιορισμοί

## Setup overhead

Η γείωση + παραγωγή facts μπορεί να καθυστερήσει απλές εντολές.

## Debugging όταν δεν υπάρχει λύση

SAT-style εξηγήσεις δεν είναι πάντα «φιλικές» στον χρήστη.

## Προοπτικές

- βελτιστοποίηση setup phase
- incremental solving
- εφαρμογή ιδέας σε cloud orchestration/microservices

## Takeaway

Ο ASP μετατρέπει την επίλυση εξαρτήσεων από ευριστικό «κόλπο» σε μαθηματικά ελεγχόμενη διαδικασία.

Ευχαριστούμε για την προσοχή σας!