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

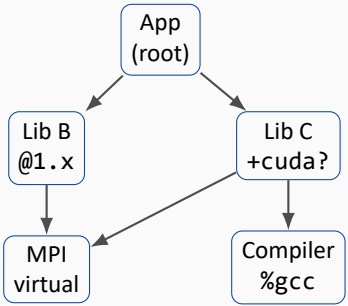
HPC Stack

- **Εκδόσεις**: πολλαπλές εκδόσεις συνυπάρχουν
- **Compilers**: GCC ⇒ 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

System package managers (APT/RPM)

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

Language managers (pip/cargo)

- συχνά αγνοούν system deps (C/C++, compilers)
- ad-hoc επίλυση

Το παλιό Spack (greedy)

- αποφάσεις «χωρίς επιστροφή» (no backtracking)
- **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

Συμπληρώνοντας τα κενά με constraints έτσι ώστε: (i) να ικανοποιούνται περιορισμοί χρήστη/πακέτων/περιβάλλοντος, και (ii) να μην υπάρχουν conflicts.

ASP Implementation

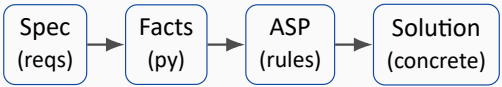
Declarative μοντέλο

- **Facts**: δεδομένα (available versions, deps)
- **Rules**: παραγωγή γνώσης (propagation)
- **Constraints**: τι απαγορεύεται

Ροή

- **Python** παράγει facts
- **Grounding** $\rightarrow$ προτασιακό πρόγραμμα
- **clingo** $\rightarrow$ stable models (stable models)

Παράδειγμα pipeline



Γιατί όχι καθαρό SAT;

Το μοντέλο HPC είναι εκφραστικά «βαρύ». Ο ASP δίνει πιο φυσικούς/συντηρήσιμους κανόνες.

Ενδεικτικά facts (όπως τα παράγει το Spack)

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”).
```

Σχόλιο

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

Κανόνας επιλογής: «μία έκδοση ανά κόμβο»

Choice rule (πυρήνας)

```
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.

Διάδοση εξαρτήσεων: «αν υπάρχει πακέτο, υπάρχουν και οι deps»

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 **πληρότητα**: αν υπάρχει λύση, θα βρεθεί.

Virtual dependencies (virtuals): επιλογή παρόχου

Ιδέα

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

Κανόνας (σχηματικά)

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

Παράδειγμα

- hdf5 depends on mpi
- providers: mpich, openmpi
- επιλογή βάσει preferences (preferences) + συγκρούσεων

Σημείο-κλειδί

Η επιλογή provider είναι μέρος της ίδιας ενιαίας βελτιστοποίησης.

Conditional deps (conditionals): όταν μια εξάρτηση εξαρτάται από variant

Πρόβλημα (greedy)

Πρέπει να «μαντέψεις» πρώτα το `+bzip2` ή όχι, αλλιώς μπορεί να πέσεις σε αδιέξοδο.

ASP (γενικευμένες συνθήκες)

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

```
condition_holds(ID):-node(P),variant_
value(P,bzip2,true).dependency_enabled(P,
bzip2):-condition_holds(ID).
```

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 πολυκριτηριακή βελτιστοποίηση

Ιδέα

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

Ενδεικτικά κριτήρια (υψηλή προτεραιότητα)

- **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** ↓ (ABI)
- **Target mismatch** ↓

Καινοτομία: software reuse (reuse) + buildcache (buildcache)

Το «κόλπο» με τους κάδους

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

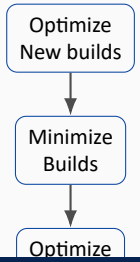
Λογική πολιτική

«Αν πρέπει να χτίσεις, χτίσε το τέλειο. Αν υπάρχει συμβατό binary, προτίμησέ το για να γλιτώσεις χρόνο».

Γιατί έχει σημασία

- builds στο HPC είναι ακριβά (χρόνος/πόροι)
- reuse = γρηγορότερα installs ⇒ καλύτερο UX

Σχηματικό



Απόδοση στην πράξη (E4S και πραγματικά HPC συστήματα)

Περιβάλλον

- E4S repository (χιλιάδες πακέτα)
- Quartz (Intel Xeon) και Lassen (Power9 + NVIDIA GPU)

Κλιμάκωση με reuse

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

Κύρια ευρήματα

- **Solve times**: συνήθως $< \mathbf{1}$ sec
- χρόνος $\uparrow$ με πολλές εναλλακτικές εξαρτήσεις (choices)
- ρύθμιση clingo **tweety** καλύτερη (έναντι trendy/handy)

Πού είναι το bottleneck;

Μετατοπίζεται στην **Python setup phase** (εξαγωγή facts/grounding), όχι στο ίδιο το solving.

Ποιότητα λύσεων: το τέλος των false negatives

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 / τι μένει δύσκολο

Setup overhead

Η γείωση + παραγωγή facts (Python) μπορεί να καθυστερήσει απλές εντολές.

Debugging όταν δεν υπάρχει λύση

SAT-style εξηγήσεις (unsat cores) δεν είναι πάντα «φιλικές» στον χρήστη.

Προοπτικές

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

Takeaway (1 πρόταση)

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

Συμπέρασμα (για KRR)

Πού «κουμπώνει» στο μάθημα;

- Αναπαράσταση γνώσης: facts/rules/constraints ως μοντέλο του οικοσυστήματος πακέτων
- Αυτόματη συλλογιστική: stable models ως λύσεις του concretization
- Βελτιστοποίηση: τυπικά κριτήρια, όχι ad-hoc heuristics

Μήνυμα

Όταν το πρόβλημα είναι συνδυαστικό και γεμάτο περιορισμούς, το **να το γράψεις ως λογική** μπορεί να είναι πιο πρακτικό από το να «μπαλώσεις» heuristics.

Ευχαριστούμε για την προσοχή σας!

Πηγή: Gamblin et al. (2022) — `paper.pdf` (με highlights σε σελ. 4–12)