



# Budowanie i Instalacja Aplikacji na Systemach HPC

Jakub Gałecki, jgalecki@icm.edu.pl

29.01.2024

# Sylwetka prowadzącego

Członek Zespołu Oprogramowania i Wsparcia Użytkowników ICM

Praca naukowa dotyczy Metody Elementów Skończonych w zastosowaniach związanych z mechaniką płynów

Jeden z maintainerów stosu aplikacji ICM

# Agenda

- Motywacja
- Z czego składa się zbudowana aplikacja?
- Kroki budowania aplikacji
  - Konfiguracja
  - Kompilacja
  - Linkowanie
  - Instalacja
- Narzędzia
  - Make
  - GNU Autotools
  - CMake
  - Spack*
- Sesja praktyczna

# Motywacja

Grupa badaczy lub developerów napisała oprogramowanie, służące do Obliczeń Ciekawych Zjawisk Fizycznych i udostępniła je jako repozytorium na Githubie. Chcielibyśmy użyć tego narzędzia na klastrze obliczeniowym [ICM].

# Motywacja

Grupa badaczy lub developerów napisała oprogramowanie, służące do Obliczeń Ciekawych Zjawisk Fizycznych i udostępniła je jako repozytorium na Githubie. Chcielibyśmy użyć tego narzędzia na klastrze obliczeniowym [ICM].

# Repozytorium

- Kod źródłowy
- **Dokumentacja**
  - User's guide
  - Instalacja
  - FAQ
- Przykłady
- Skrypty
- Issues – dyskusja
- ...

# Anatomia aplikacji

- Plik wykonywalny (lub kilka)
- Biblioteki
- Nagłówki
- Dokumentacja
- Dane
- Pliki ułatwiające konsumpcję

# Anatomia aplikacji

- Plik wykonywalny (lub kilka)
- Biblioteki
- Nagłówki
- Dokumentacja
- Dane
- Pliki ułatwiające konsumpcję

Efekt finalny:

```
srun moja_aplikacja plik_wsadowy
```

# Anatomia aplikacji

- Plik wykonywalny (lub kilka)
- Biblioteki
- Nagłówki
- Dokumentacja
- Dane
- Pliki ułatwiające konsumpcję

Efekt finalny:

```
srun moja_aplikacja plik_wsadowy
```

lub konsumpcja jako biblioteka przez inną aplikację

# O co tyle hałasu?

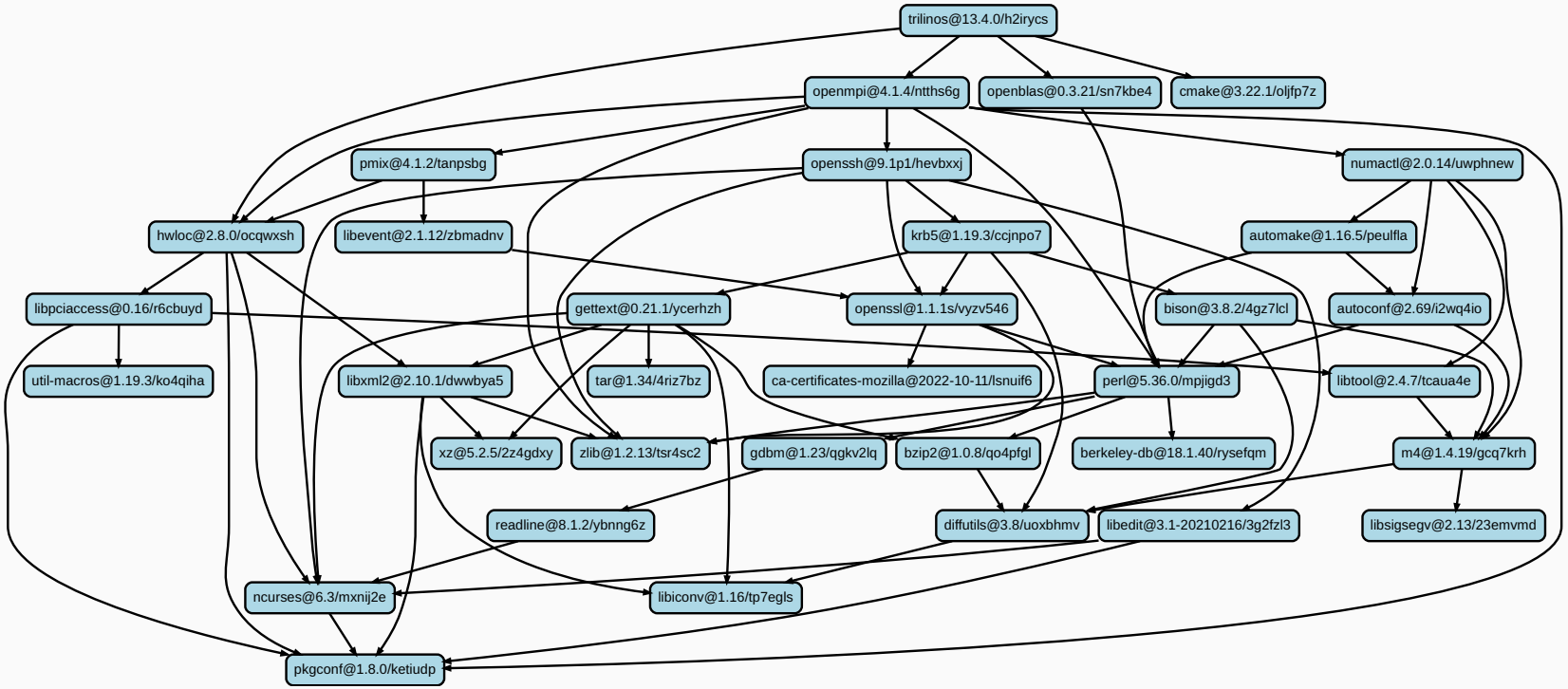
Instalacja np. przeglądarki na prywatnych komputerach polega na ściągnięciu instalatora i kliknięciu go 2 razy myszą. Czemu nie możemy zrobić tego samego na KDM?

# O co tyle hałasu?

Instalacja np. przeglądarki na prywatnych komputerach polega na ściągnięciu instalatora i kliknięciu go 2 razy myszą. Czemu nie możemy zrobić tego samego na KDM?

- Dependencje 🐉
- Wykorzystanie dostępnych zasobów sprzętowych, np. akceleratorów
- Optymalna konfiguracja
- Brak `sudo`

# Przykładowy graf dependencji



# Budowanie aplikacji

# Konfiguracja

- Wykrywanie kompilatora i linkera
- Identyfikacja potrzebnych flag
- Wyszukiwanie dependencji
- Generacja kodu
- Generacja plików wsadowych do systemu budowania (Makefile etc.)
- Ściąganie kodu i budowanie dependencji

Owocem konfiguracji jest przepis* na kompilację i linkowanie aplikacji

```
./configure CC=mpicc
```

```
cmake -DCMAKE_C_COMPILER=mpicc ..
```

# Makefile

Na systemach Unix, powszechnym formatem “przepisu” na aplikację jest `Makefile`.

Narzędziem, którym go wywołujemy jest `make`.

Możemy podawać różne *targety*, np.:

- `clean` czyści katalog
- `check` uruchamia testy
- `install` instaluje aplikację

Dodatkowo należy pamiętać o fladze `-j`, dzięki której możemy zrównoleglić proces budowania.

```
make -j 8
make check
make install
```

# Kompilacja

Generacja plików zawierających kod obiektowy dla każdej z jednostek translacji (plików źródłowych), na podstawie kodu źródłowego.

Wykorzystane w danym TU funkcje (i nie tylko), których definicje znajdują się w innym TU muszą mieć widoczne *deklaracje*.

Jednostki translacji są od siebie w pełni niezależne.

Na tym etapie kompilator ma największe pole do optymalizacji.

```
gcc -c -o out.o in.cpp
```

# Linkowanie 

Generacja pliku wykonywalnego lub biblioteki na podstawie plików obiektowych (i opcjonalnie istniejących bibliotek).

Potencjalnie możliwość optymalizacji (LTO).

W zamyśle prosty krok, ale możemy natknąć się na różne problemy:

- Brak biblioteki*
- Brak symboli w bibliotece
- Niekompatybilność wersji
- Niekompatybilność interfejsu binarnego (ABI) 

```
gcc main.o obj1.o obj2.o
```

# Przydatne flagi

`-I/sciezka/do/naglowka` – wskazuje kompilatorowi, gdzie szukać plików nagłówkowych

`-L/sciezka/do/biblioteki` – wskazuje linkerowi, gdzie szukać bibliotek

`-lbiblioteka` – wskazuje linkerowi, jakie biblioteki linkować

Niestety czasem musimy podawać je ręcznie...

# Linkowanie statyczne vs. dynamiczne

Biblioteka statyczna – efektywnie archiwum skompilowanych wcześniej plików obiektowych. Na systemach Unixowych wg. konwencji ma rozszerzenie `.a`.

Linkowanie statyczne przebiega efektywnie tak, jakby biblioteka była częścią projektu, który budujemy

# Linkowanie statyczne vs. dynamiczne

Biblioteka statyczna – efektywnie archiwum skompilowanych wcześniej plików obiektowych. Na systemach Unixowych wg. konwencji ma rozszerzenie `.a`.

Linkowanie statyczne przebiega efektywnie tak, jakby biblioteka była częścią projektu, który budujemy

Biblioteka dynamiczna – ładowana przez program *na początku jego uruchomienia* (program loader)

Linkowanie dynamiczne polega na odpowiednim zakodowaniu w pliku wykonywalnym, jakie biblioteki mają być odszukane przy ładowaniu programu

`LD_LIBRARY_PATH` vs. `LD_RUN_PATH`

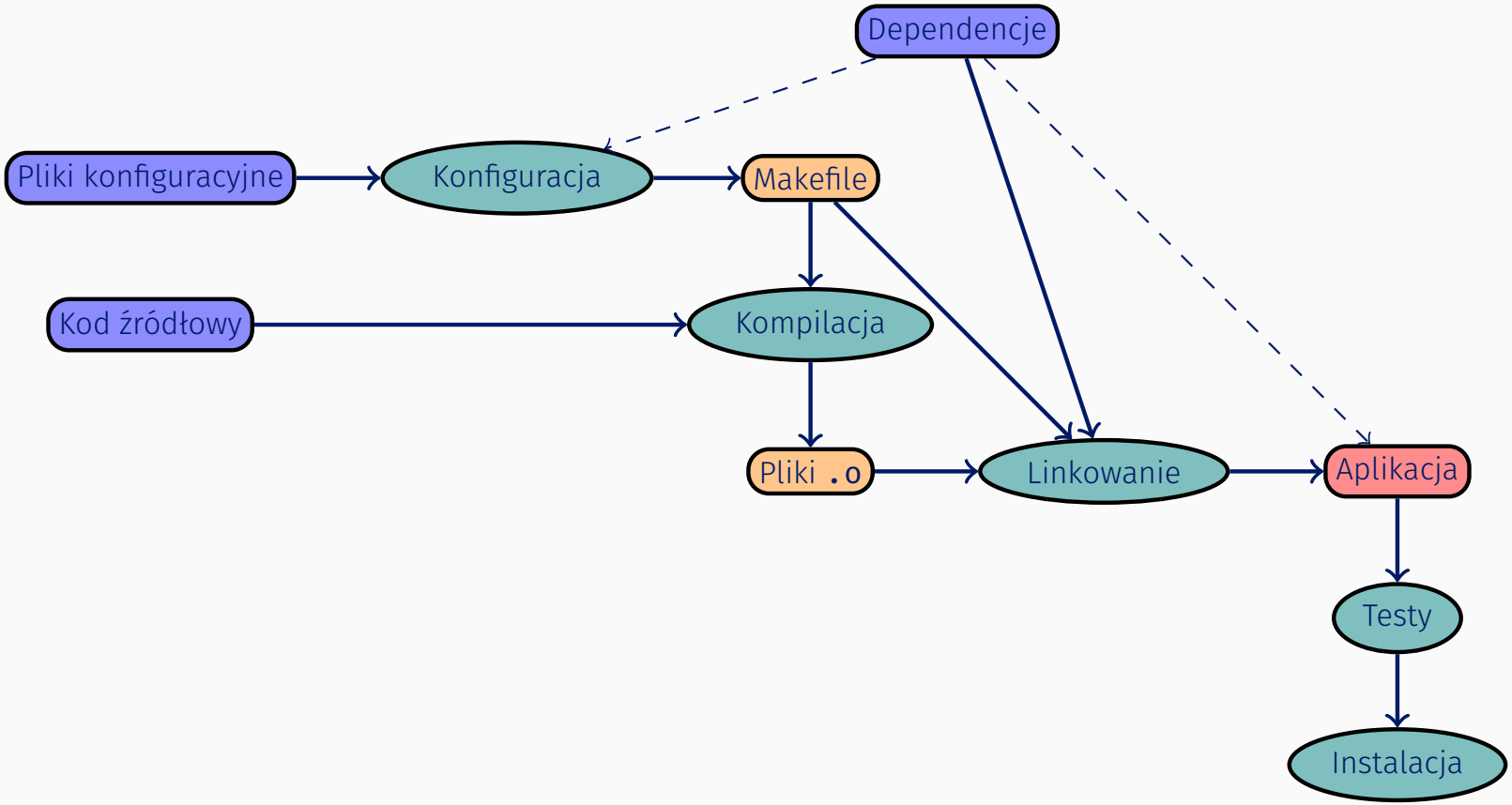
# Instalacja

Najprostszy krok – kopiowanie utworzonych plików wykonywalnych i bibliotek do docelowych katalogów.

Ewentualnie konfiguracja środowiska/specjalnych plików celem ułatwienia znalezienia zainstalowanej aplikacji.

**Przed zainstalowaniem należy przetestować zbudowaną aplikację!**

# Budowanie aplikacji



# GNU Autotools

Autotools: Autoconf, Automake, Libtool, M4

Zestaw niskopoziomowych narzędzi pomagających tworzyć skrypt konfiguracyjny aplikacji

Potrzebne do budowania wielu aplikacji/bibliotek HPC

W opinii autora: należy unikać

# Make (+ Autotools) quickstart

```
# Zaczynamy w glownym katalogu repozytorium
make configure # czesto niepotrzebne
./configure CC=mpicc                              \
  CFLAGS="-O3 -march=native -mtune=native"        \
  --prefix="sciezka/do/instalacji"                \
  --with-dependencja="sciezka/do/dependencji" \
  --enable-opcja                  || exit 1
make -j $SLURM_CPUS_ON_NODE || exit 1
make check                        || exit 1
make install
```

# Make (+ Autotools) quickstart – wersja trudniejsza

```
# Zaczynamy w glownym katalogu repozytorium
cp Makes/Makefile.arch .
vim Makefile.arch # reczna specyfikacja flag etc.
make -j $SLURM_CPUS_ON_NODE || exit 1
make check                  || exit 1
make install
```

# CMake

Zintegrowany system generowania plików Makefile (i ekwiwalentów)

Wysoki poziom abstrakcji – CMake jest językiem programowania ze zmiennymi, funkcjami, “biblioteką standardową”, etc.

Wysoki poziom standardyzacji procesu budowania

Łatwe i przejrzyste zarządzanie grafem dependencji

Wbudowany framework do testów (CTest)

Rozsądna dokumentacja

# CMake quickstart

```
# Zaczynamy w glownym katalogu repozytorium
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release                    \
  -DCMAKE_C_COMPILER=mpicc                          \
  -DCMAKE_C_FLAGS="-march=native -mtune=native"     \
  -DCMAKE_INSTALL_PREFIX="sciezka/do/instalacji"    \
  -DDependencja_DIR="sciezka/do/dependencji"        \
  -DOPCJA_BIBLIOTEKI=ON                             \
  ..                                          || exit 1
cmake --build . -- -j $SLURM_CPUS_ON_NODE     || exit 1
ctest --output-on-failure                     || exit 1
cmake --install .
```

# Na maszynach ICM

Kompilator (przeważnie wrapper MPI) ładujemy z modułu

Na topoli: OpenMPI 4.1.6

Wiele dependencji dostępne przez moduł (BLAS, Scalapack, ...)

Te same moduły powinny być załadowane przy budowaniu i uruchamianiu programu

# Dziękuję za uwagę

## Pytania?

jgalecki@icm.edu.pl