BME TMIT
2022

14/12

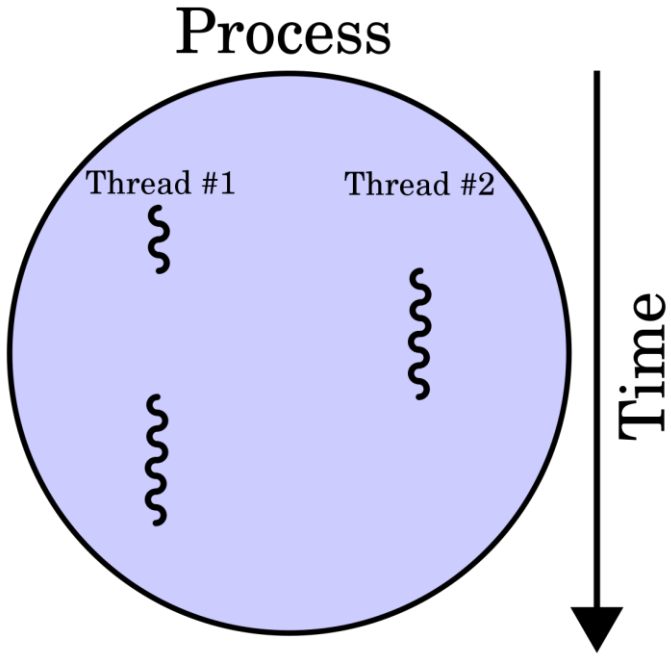
Németh Gábor
Janky Ferenc Nándor előadásvázlatai alapján

Funkcionális programozás C++-ban

Párhuzamosság és funkcionális programozás

C++ program végrehajtás I.

- A program a végrehajtás szempontjából lehet
  - egyszálú
  - többszálú, párhuzamos végrehajtás
- Időben a végrehajtás szálainak száma változhat
  - pl.: tipikusan 1 szál program indulásánál és 1 szál a futás végén
- Ha több szoftver szál fut mint ahány hardveres végrehajtási egység elérhető, nem feltétlenül nő a kihasználtság
- A CPU által fizikailag egyszerre futtatható szálak az `std::thread::hardware_concurrency()` könyvtári függvénnyel kérhető le



https://en.wikipedia.org/wiki/File:Multithreaded_process.svg

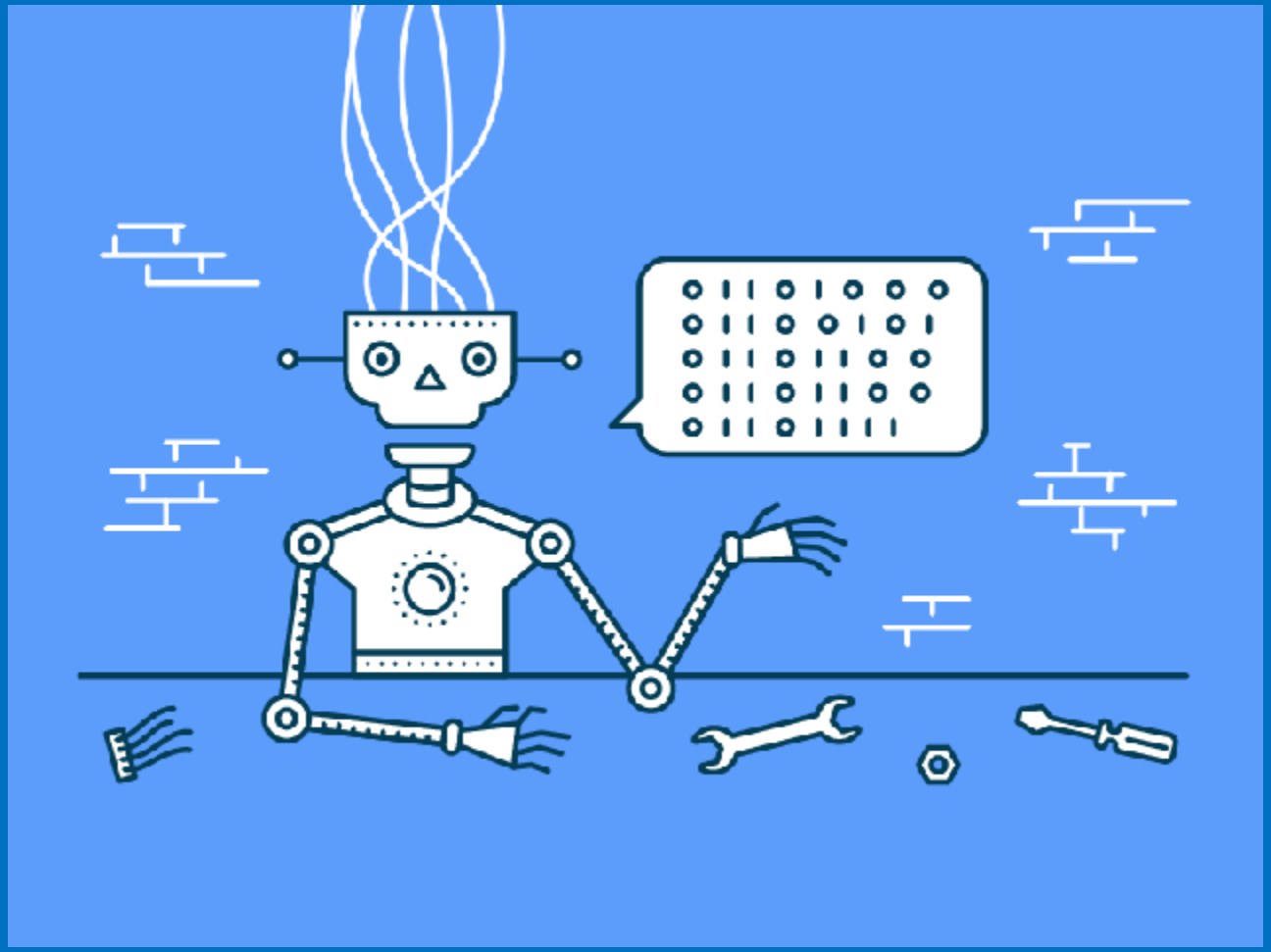
C++ program végrehajtás II.

- Kötelező párhuzamosság
  - Kötelező párhuzamosság esetén explicit meghatározzuk, hogy hány szálon hajtódik végre a programunk, hogyan történik a szálak közötti kommunikáció, szinkronizáció, stb.
  - Ha megváltozik a HW környezet, akár még rosszabbul is működhet

- Opcionális párhuzamosság
  - Opcionális párhuzamosság esetén az elvégzendő feladatot határozzuk meg, és egy runtime-ra bízzuk az optimális végrehajtást
  - Ha megváltozik a HW környezet, a program újrafordítás nélkül is képes kiaknázni pl. a megnövekedett számú végrehajtási egységeket

- **Törekedjünk az opcionális párhuzamosságra!**

ex_0: kötelező- és opcionális párhuzamosság



Párhuzamos végrehajtás problémái

- Szinkronizáció szükséges a végrehajtási szálak közötti adatmanipulációhoz
- Konkurens adatszerkezetek alkalmasak szinkronizációra
  - *Locking*
  - *Lock-free*
  - *Wait-free*

- Szinkronizáció nélkül versenyhelyzet => *Undefined Behaviour (UB)*
  - *std::atomic<>* - jól definiált hozzáférési sorrend, megfelelően címkézett írások/olvasások esetén nincs *UB*

Párhuzamosság

- Miért is törekszünk a párhuzamosságra, ha ilyen sok a nehézség és a potenciális komplikáció?
  - párhuzamossággal szeparáljuk a különböző egységeket/problémákat
    - pl.: nem szeretjük, ha a UI nem reszponzív, amíg egy vagy több háttérfolyamat zajlik
  - párhuzamossággal növeljük a szoftverünk teljesítményét
    - elértünk oda, hogy a CPU-k órajelnövekedése, már nem kifizetődő a szoftver teljesítmény szempontjából. A trend az, hogy egyre több párhuzamos végrehajtó egységet integrálnak a CPU-kba

Párhuzamosság és funkcionális programozás

- Miből fakadnak a problémák?
- Ha nincs ütközés => nincs versenyhelyzet
- Funkcionális programban tiszta függvényeket használunk:
  - nincsenek írható globális változók
  - nincsnek mellékhatások
- Tiszta függvényeket tetszőleges párhuzamosíthatjuk, mivel definíció szerint nincs versenyhelyet

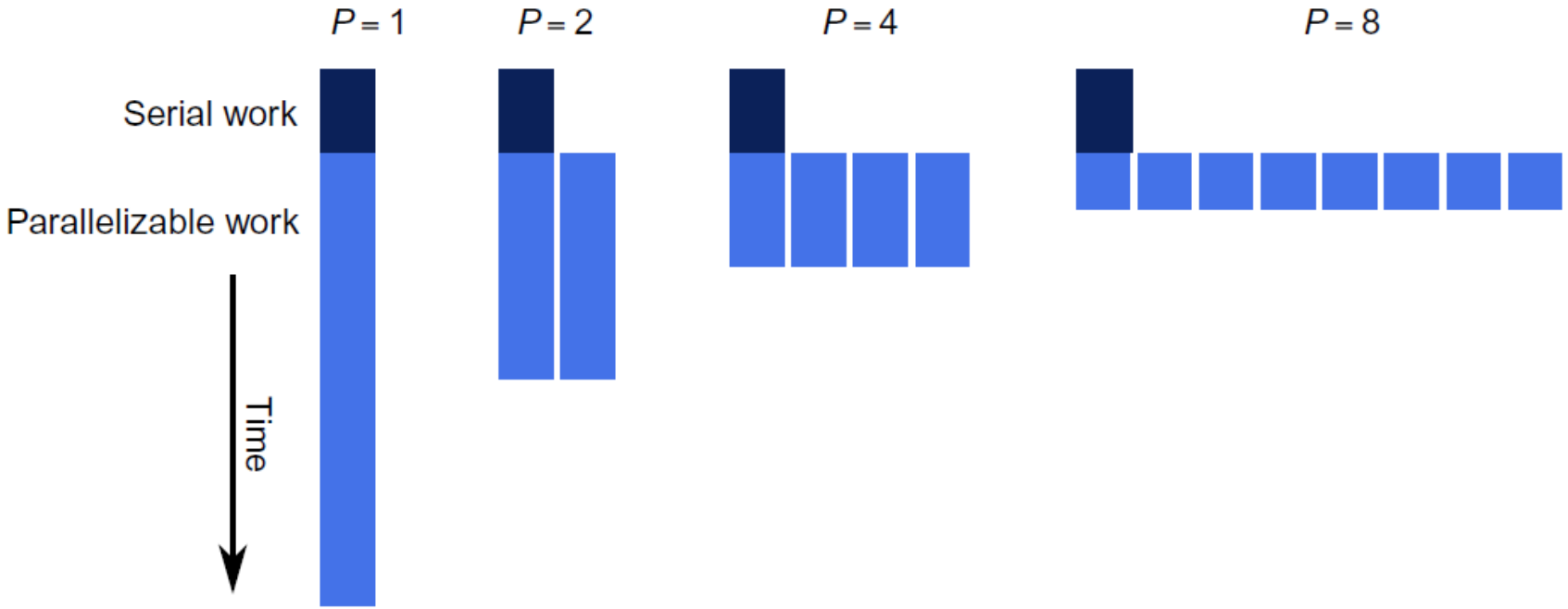
Párhuzamosság és gyorsulás elvi határa

- **gyorsulás:** $s_P = \frac{T_1}{T_P}$ (1)
  - ahol, T_1 az egyszálú program futási ideje
  - T_P a P szálon végrehajtott program futási ideje
- **hatékonyság:** $E_P = \frac{S_P}{P} = \frac{T_1}{P \times T_P}$ (2)
  - ha a program P szálon P-szer gyorsabban fut, akkor $S_P = P$, ekkor (1)-ből $T_P = \frac{T_1}{P}$, (2) –be helyettesítve $E_P = 1$, **lineáris gyorsulás**
  - gyakorlatban $E_P < 1$, mivel a többszálú feladatkezelésnek *overhead*-je van
  - létezik $E_P > 1$, **szuperlineáris gyorsulás**, ahol a párhuzamos program jobban kihasználja a rendelkezésre álló hardver erőforrásokat

Amdahl törvénye I.

- Gene Amdhal 1967-ben prezentálta
- T_1 felbontható két részre
  - $T_1 = W_{ser} + W_{par}$
  - Egy olyan időtartamra ahol, a program csak soros munkát végez, és egy olyanra, ahol párhuzamosítható munkát végez
- T_P-re az alábbi alsó korlátot adta
  - $T_P \geq W_{ser} + \frac{W_{par}}{P}$
- Behelyettesítve (1)-be egy felső korlátot kapunk a gyorsulásra:
  - $S_p \leq \frac{W_{ser}+W_{par}}{W_{ser}+\frac{W_{par}}{P}}$ (3)

Amdahl törvénye II.

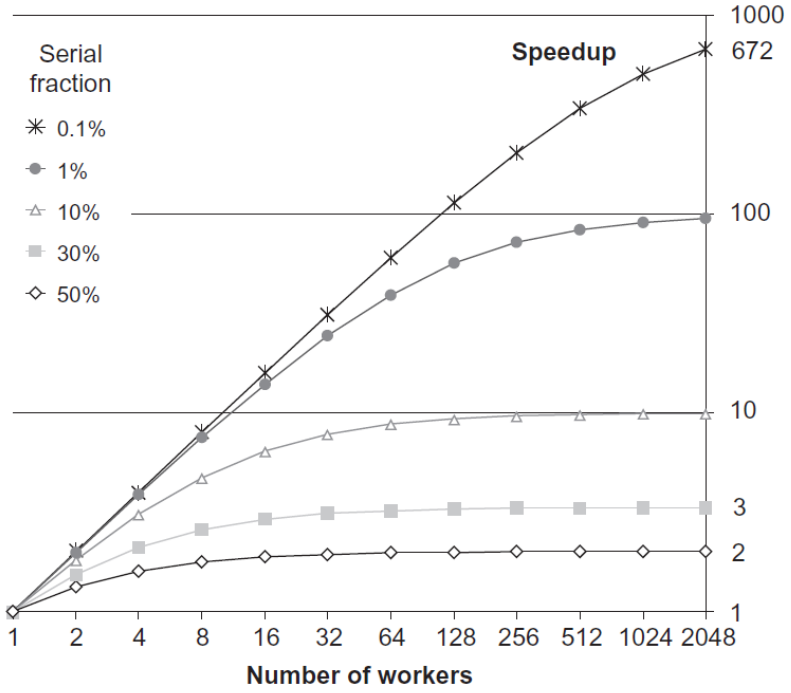


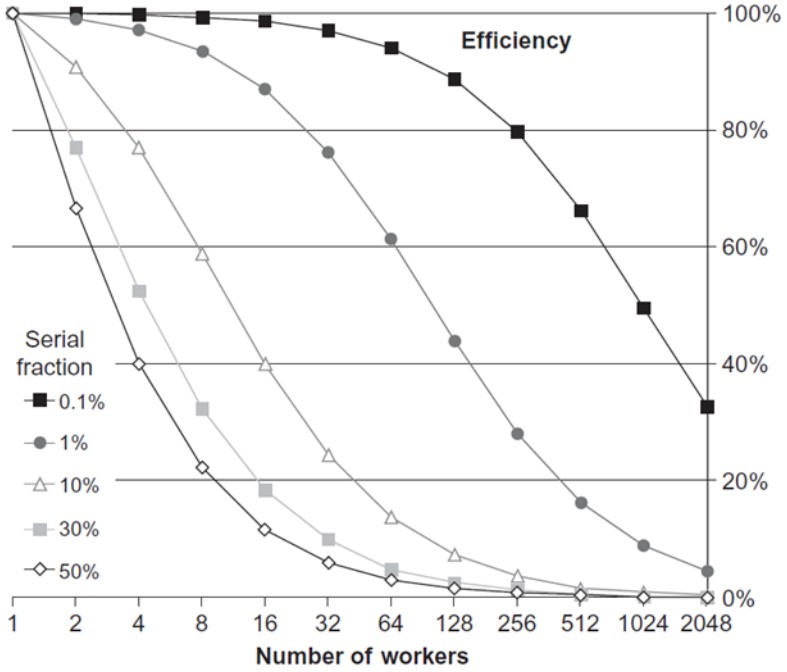
forrás: Structured Parallel Programming,Patterns for Efficent Computation, Figure 2.4

Amdahl törvénye III.

- Ha f a nem párhuzamosítható hányada a program futásának, akkor
  - $W_{ser} = f \times T_1$
  - $W_{par} = (1 - f) \times T_1$
- Behelyettesítve (3)-ba
  - $S_p \leq \frac{1}{\mathrm{f}+\frac{1-\mathrm{f}}{\mathrm{P}}}$ (3)
- Mi történik, ha P-t minden határon túl növeljük?
  - $S_\infty \leq \frac{1}{f}$

Amdahl törvénye IV.



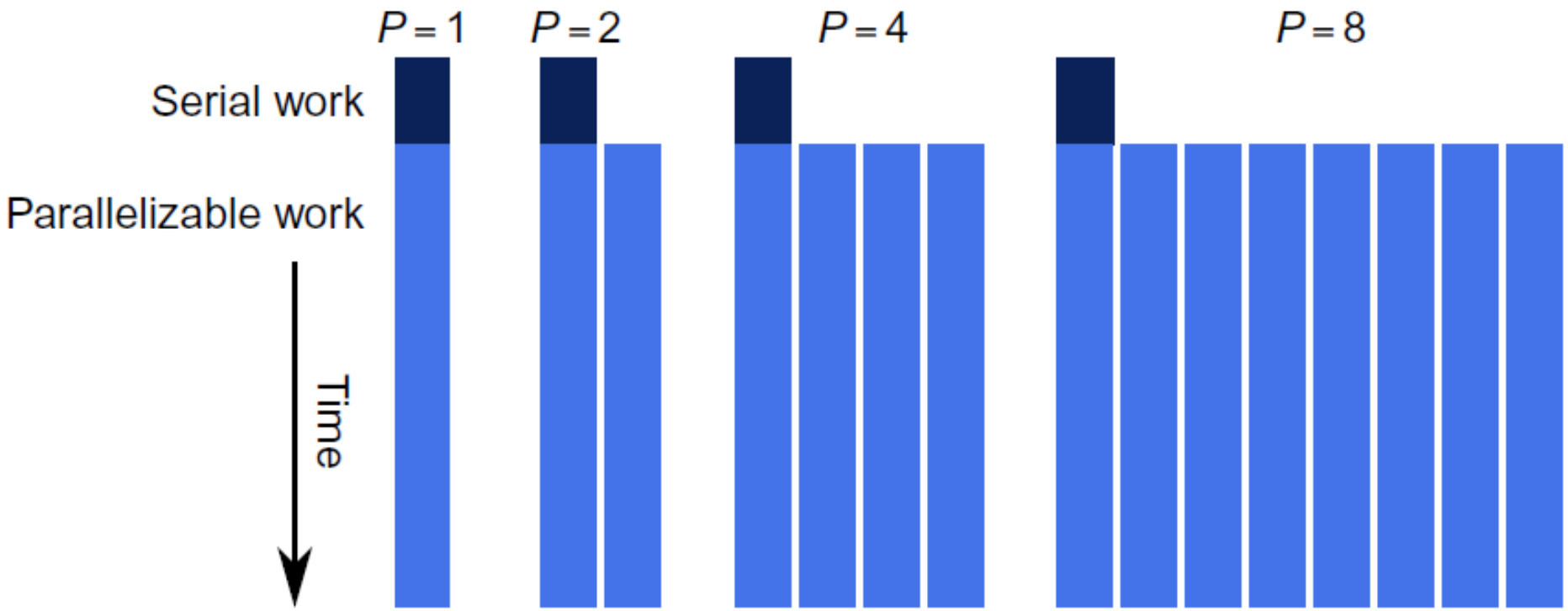


forrás: Structured Parallel Programming,Patterns for Efficent Computation, Fig. 2.5 és Fig. 2.6

Gustafson törvénye I.

- John Gustafson kb. 2 évtizeddel Amdahl törvénye után észrevette, hogy a Sandia National Labs-nél a programok több, mint 1000-szeresen gyorsultak
- Amdahl rögzítette a probléma méretét, és végrehajtás idejét a párhuzamos végrehajtási szálak függvényében vizsgálta
- Gustafson megmutatta, hogy ahogy a számítógépek fejlődnek úgy a problémák méretei is növekszenek. Ahogy a probléma mérete növekszik, ha a párhuzamosítható hányad sokkal jobban növekszik mint a csak sorosan végrehajtható, nagyobb a gyorsulás mértéke
- (Mindkettőjüknek igaza volt!)

Gustafson törvénye II.



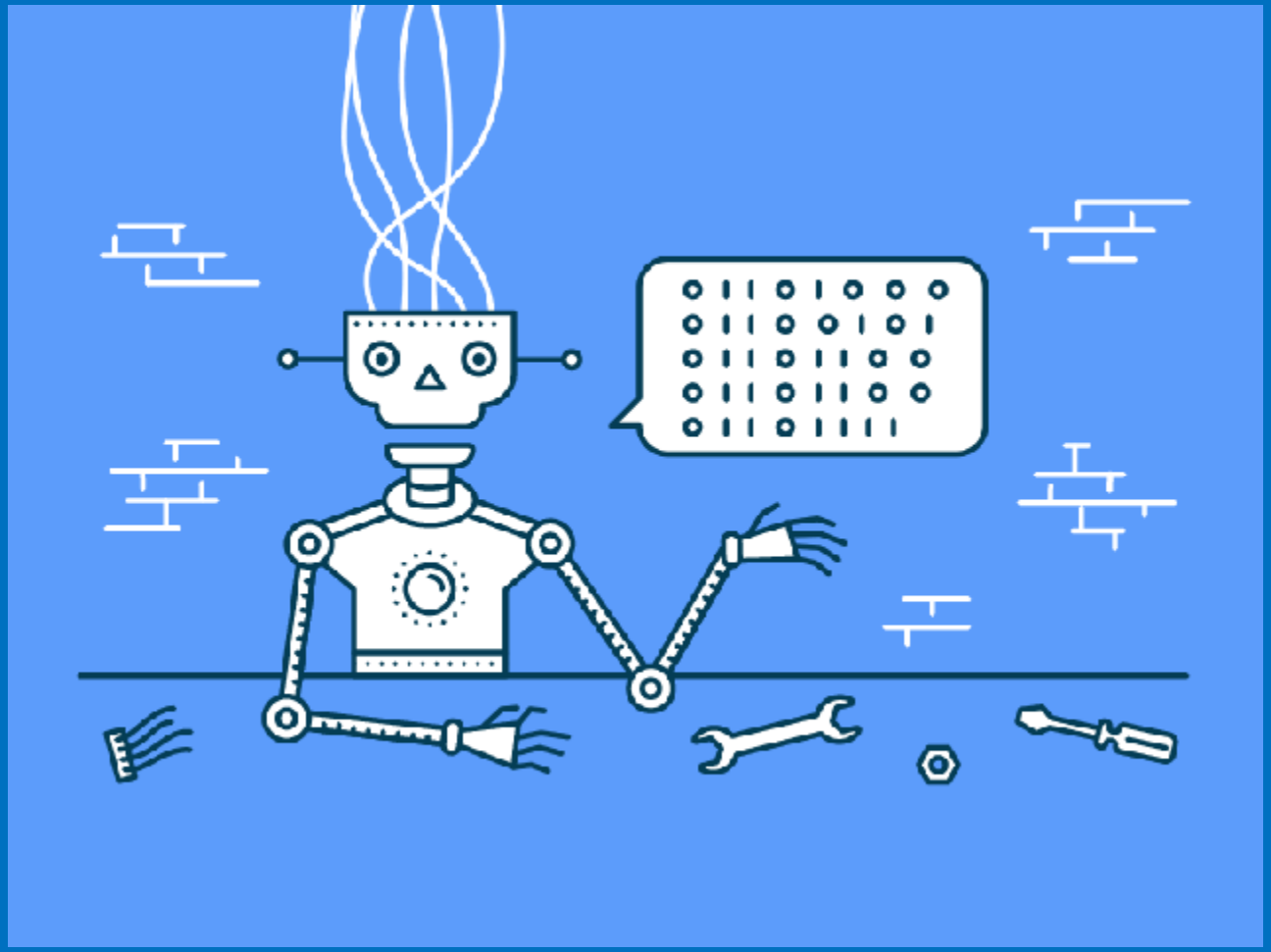
forrás: Structured Parallel Programming,Patterns for Efficent Computation, Figure 2.7

C++17 és párhuzamos algoritmusok

- C++17-től kezdődően sok algoritmus szignatúrája kiegészült egy első paraméterrel, *execution_policy* az *<execution>* fejállományból
  - *std::execution::seq* – szekvenciális végrehajtás
  - *std::execution::par* – páruzamos végrehajtás
  - *std::execution::par_unseq* – párhuzamos, vektorizált végrehajtás
  - *(C++20) std::execution::unseq* – vektorizált végrehajtás

- Az optimális végrehajtási szálak számát a runtime kezeli (opcionális párhuzamosság!)
- **A szinkronizáció továbbra is a felhasználó feladata!**

ex_1: párhuzamos algoritmusok



Összefoglaló a párhuzamosításról

- Törekedjünk az opcionális/skálázható párhuzamosságra
- Ökölszabályok
  - A rendelkezésre álló párhuzamosítást mindig a probléma méretének függvényében használjuk ki
  - Túlzott dekompozícióval védekezhetünk az egyenlőtlen terhelés ellen
  - Minimalizáljuk a szinkronizációt szükségességét (atomikus változók is szinkronizálnak!!!)
  - Ügyeljünk az időbeli és a térbeli lokalitásra, hogy minimalizáljuk a memória forgalmat
    - **Temporális lokalitás:** valószínűbb, hogy a processzor ugyanahhoz a lokációhoz fér hozzá a közeljövőben
    - **Térbeli lokalitás:** valószínűbb, hogy a processzor egy szomszédos/közeli a lokációhoz fér hozzá a közeljövőben

Köszönöm a figyelmet!