Multi-Objective Optimization

Non-dominated sorting GA and **Multi-objective PSO**

Marco S. Nobile, Ph.D. – marco.nobile@unimib.it

University of Milano-Bicocca, Department of Informatics, Systems and Communication

PhD program in Computer Science

Outline

- Multi-Objective Optimization (MOO)
    - Dominance
    - Pareto Optimality
    - Crowding distance
    - Non-dominated Sorting Genetic Algorithm (NSGA)
    - NSGA-II
    - NSGA-III
    - Multi-Objective PSO (MOPSO)

- High Performance Computing
    - Widespread architectures
    - GPGPU computing
    - Best practices and some future directions

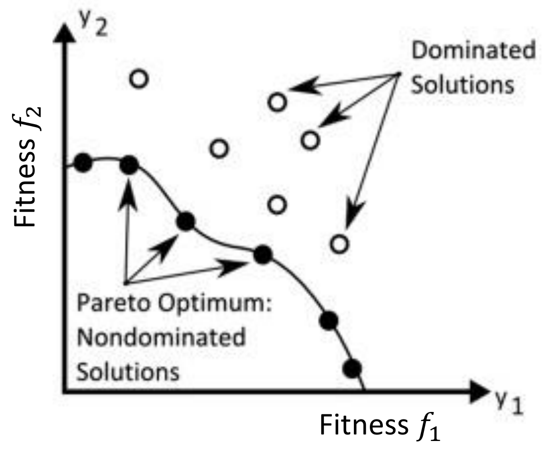
Multi-Objective Optimization (MOO)

- Classic global optimization: **single objective/criterion**
  - E.g., minimize or maximize a given fitness function $f(\mathbf{x}) \to \mathbb{R}$
  - Where $\mathbf{x} \in \mathbb{R}^D$ is a D-dimensional candidate solution
  - GA, DE, PSO, FST-PSO, etc. can solve this class of problems

- Sometimes, problems have **multiple objectives** ($k \geq 2$)
  - i.e., **multiple fitness** functions $f_1, f_2, \dots, f_k$ to be **simultaneously** optimized
  - These functions can be **conflicting** (e.g., autonomous cars: minimize time-to-destination, minimize consumption of fuel/energy) [Hans, 1988]
  - **A trivial combination** of the fitness values **does not work** (e.g., we cannot just sum up time and fuel quantity) since it generally leads to the **overfitting of a subset** of the criteria
  - Also, there might be infinite optimal (and perfectly valid) solutions: the **dominating solutions**

Domination

- A feasible solution x_1 is said to **dominate** another solution x_2 iff:
  - $f_i(x_1) \leq f_i(x_2) \;\; \forall i \in \{1,2, \dots k\}$
  - $f_j(x_1) < f_j(x_2)$ for at least one $j \in \{1,2, \dots k\}$
  - Dominance is denoted by $x_1 \prec x_2$
  - A solution x^* is called **Pareto-optimal** if $x^* \prec s$ for any other solution s in solution space $\mathcal{S}$

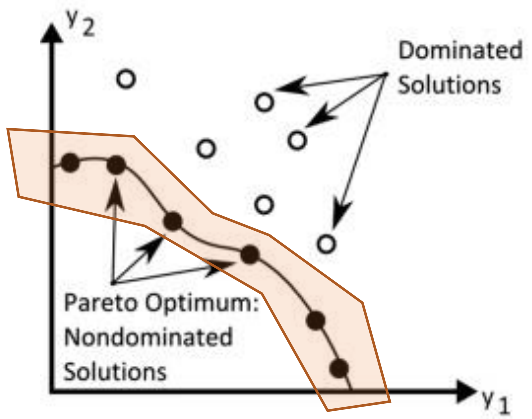


Pareto front

- In MOO, we are not interested in a *single* optimal solution
  - On the contrary, we look for the set **(front) of Pareto-dominant solutions**
  - Real-valued problems: this set clearly has (potentially) **infinite elements**
  - An **approximation** of the front would still be good

- Idea: use a **Computational intelligence** meta-heuristic
  - e.g., use **evolutionary** computation or **swarm intelligence**
  - The population of individuals «converges» to the Pareto front
  - **Loss of diversity is an issue**! We want the population to be **as «spread» as possible** to approximate the whole Pareto front

- Thus, the optimization method must move the randomly initialized individuals towards the Pareto front, **preventing the convergence** to a degenerate population **collapsed around a single point**



Pareto front

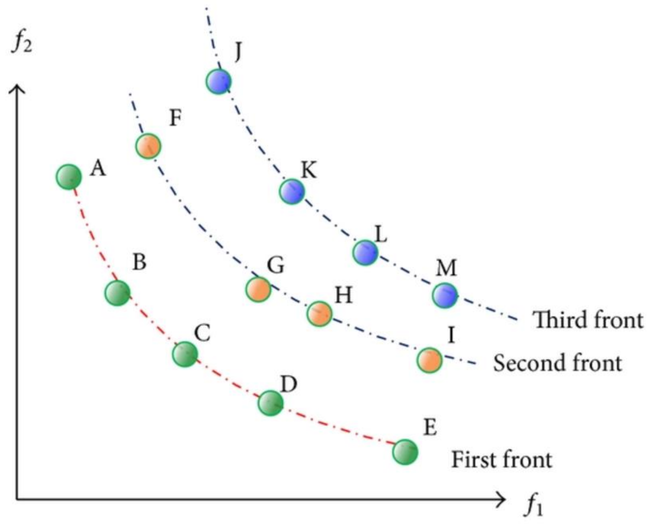
Simple (and wrong) approaches to MOO

- **Objectives weighting**: optimize $Z = \sum_{i=1}^{k} w_i f_i(\mathbf{x})$ where $0 < w_i < 1$ and $\sum_{i=1}^{k} w_i = 1$
    - Problem: the vector of weights influences the optimization. How to choose such vector?
- **Method of distance functions**: optimize $Z = \left(\sum_{i=1}^{k} |f_i(\mathbf{x}) - \overline{y_i}|^r\right)^{\frac{1}{r}}$ where $1 \leq r < \infty$
    - Generally, $r = 2$ is used
    - $\bar{\mathbf{y}} = (\overline{y_1}, \dots, \overline{y_k})$ is the vector of optima (which, sometimes, is unknown!)
    - Problem: the vector of optima influences the optimization. In particular, a wrong selection leads to non Pareto-optimal solutions
    - [Srinivas and Deb, Evol Comp 1994]
- **Vector Evaluated Genetic Algorithm** (VEGA)
    - Performs **multiple rounds of selection:** one for each objective
    - The new population is formed by using individuals selected for different objectives
    - The algorithm suffers from **speciation**: the population converges to to sub-populations that are optimal for just a single objective
    - [Schaffer, 1984]
- All these methods collapse k objective functions to a single value Z and then return a *single* optimal solution: **alternative dominating solutions** are neglected by design

Non-dominated Sorting

- Pareto-dominance induces a **partial-ordering of solutions**
    - E.g., all solutions of the Pareto-optimal front have the same rank
    - **We can sort the fronts** according to their dominance levels

- It is possible to «peel out» the fronts, one by one:
    1. $\mathcal{P}$ = population
    2. $i = 0$
    3. $i++$
    4. $\mathcal{O}$ = Pareto-optimal front of $\mathcal{P}$
    5. Save the individuals of $\mathcal{O}$ as $\mathcal{D}_i$
    6. $\mathcal{P} = \mathcal{P} \setminus \mathcal{O}$
    7. Repeat from 3 until $\mathcal{P} = \emptyset$

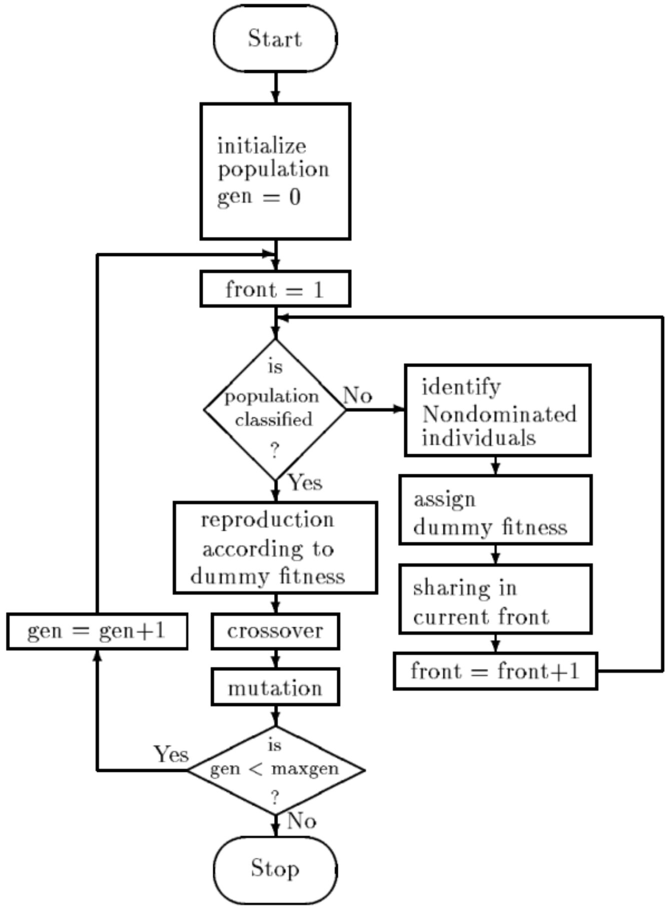


NSGA

- Similar to a classic GA except for the selection mechanism
  - Selection is based on **nondominating fronts** and **sharing**
  - A **dummy fitness**, proportional to front number, is assigned to nondominated solutions
  - **Sharing** is used to promote more diverse solutions. It considers the **phenotypic distance** d_{ij} between two individuals i and j:

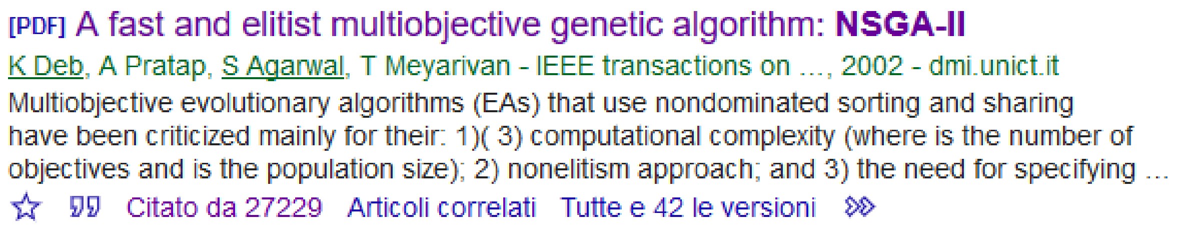
$$Sh(i,j) = \begin{cases} 1 - \left(\frac{d_{ij}}{\sigma_{share}}\right)^2 & \text{if } d_{ij} < \sigma_{share} \\ 0 & \text{otherwise} \end{cases}$$

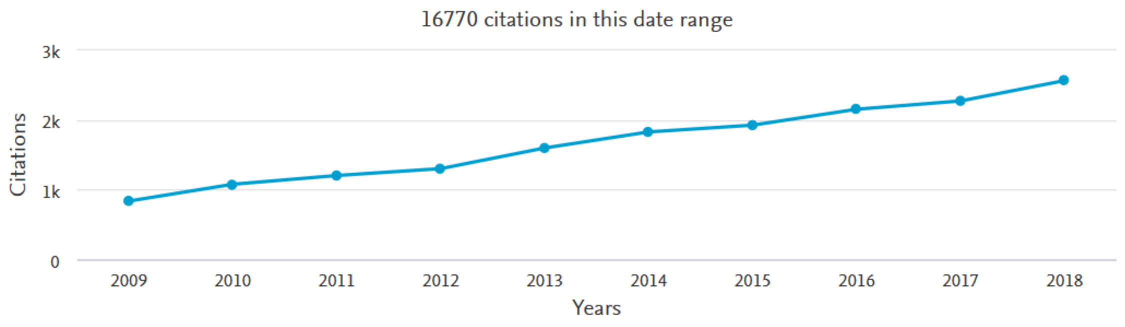
  - σ_{share} is the maximum phenotypic distance allowed between any two individuals to become members of a niche
  - [Srinivas and Deb, Evol Comp 1994]



NSGA-II

- Extends NSGA
  - Introduces **elitism**
  - Uses **fast dominated sorting** to **rank** solutions
  - Leverages **Crowding Distance** (CD) to maintain diversity in the population
  - **Tournament selection**: best ranked individuals win. If individuals have same rank, CD is used


[PDF] A fast and elitist multiobjective genetic algorithm: NSGA-II
K Deb, A Pratap, S Agarwal, T Meyarivan - IEEE transactions on ..., 2002 - dmi.unict.it
Multiobjective evolutionary algorithms (EAs) that use nondominated sorting and sharing have been criticized mainly for their: 1)(3) computational complexity (where is the number of objectives and is the population size); 2) nonelitism approach; and 3) the need for specifying ...
Citato da 27229 Articoli correlati Tutte e 42 le versioni

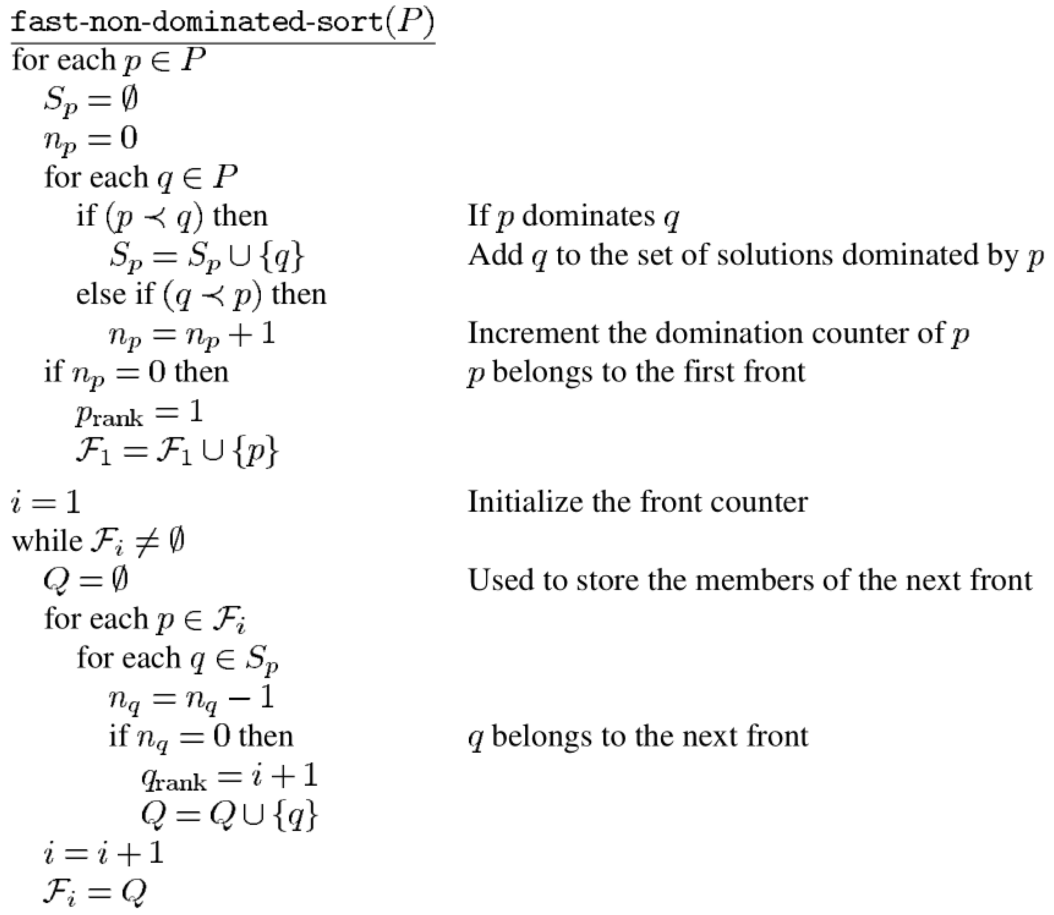


Domination ranking

- **Naïve identification of Pareto fronts**:
  - Each solution must be compared to any other solution, checking all k objectives, to determine domination: $\mathcal{O}(k \cdot N)$
  - The process must be repeated for the remaining individuals in the front, leading to a complexity of $\mathcal{O}(k \cdot N^2)$ for the identification of the *first* Pareto front
  - In the worst case, we need to repeat that process for N fronts, giving a final complexity of $\mathcal{O}(k \cdot N^3)$
  - Spatial complexity is $\Theta(N)$

NSGA-II’s fast dominated sorting

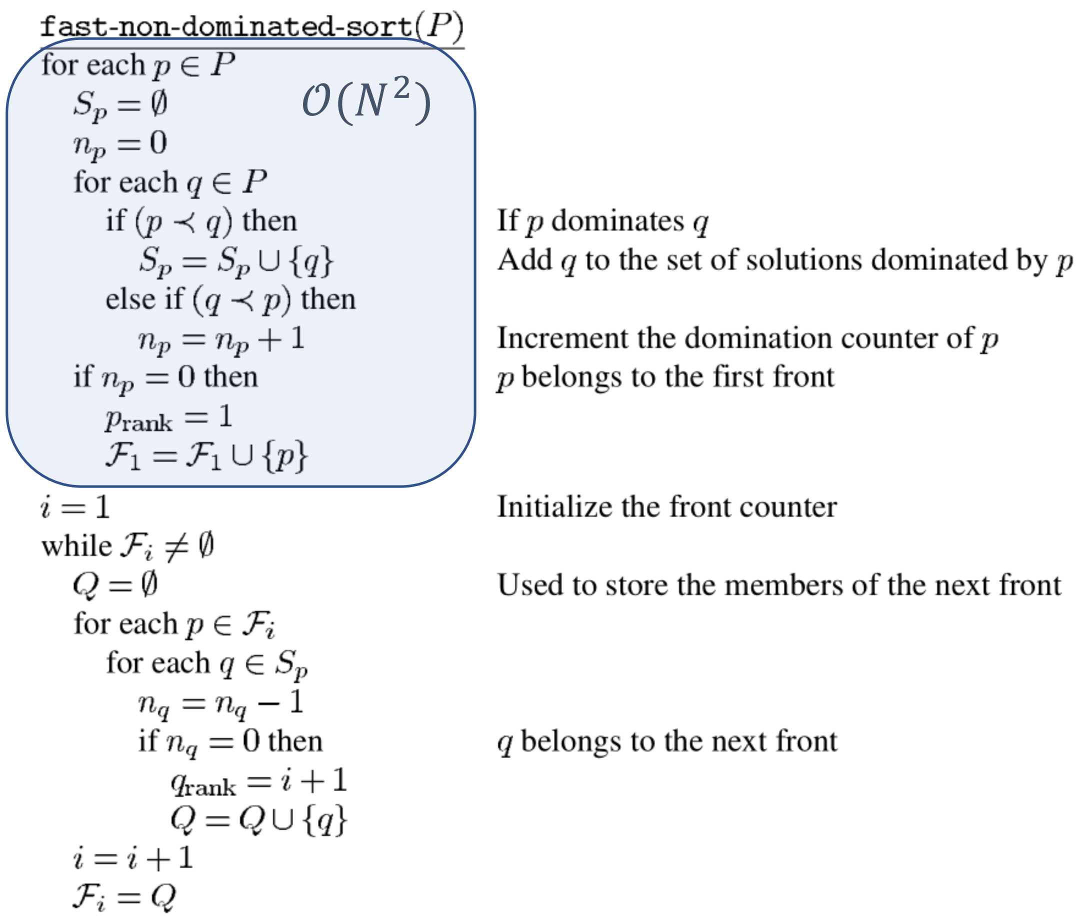
- Time complexity reduced to $\mathcal{O}(k \cdot N^2)$
- Spatial complexity increased to $\mathcal{O}(N^2)$
- [Deb *et al.*, IEEE Tran Evol Comp 2002]



```
fast-non-dominated-sort($P$)
for each $p \in P$
    $S_p = \emptyset$
    $n_p = 0$
    for each $q \in P$
        if $(p \prec q)$ then                  If $p$ dominates $q$
            $S_p = S_p \cup \{q\}$             Add $q$ to the set of solutions dominated by $p$
        else if $(q \prec p)$ then
            $n_p = n_p + 1$                    Increment the domination counter of $p$
    if $n_p = 0$ then                          $p$ belongs to the first front
        $p_{\text{rank}} = 1$
        $\mathcal{F}_1 = \mathcal{F}_1 \cup \{p\}$
$i = 1$                                        Initialize the front counter
while $\mathcal{F}_i \neq \emptyset$
    $Q = \emptyset$                            Used to store the members of the next front
    for each $p \in \mathcal{F}_i$
        for each $q \in S_p$
            $n_q = n_q - 1$
            if $n_q = 0$ then                  $q$ belongs to the next front
                $q_{\text{rank}} = i + 1$
                $Q = Q \cup \{q\}$
    $i = i + 1$
    $\mathcal{F}_i = Q$
```

NSGA-II’s fast dominated sorting

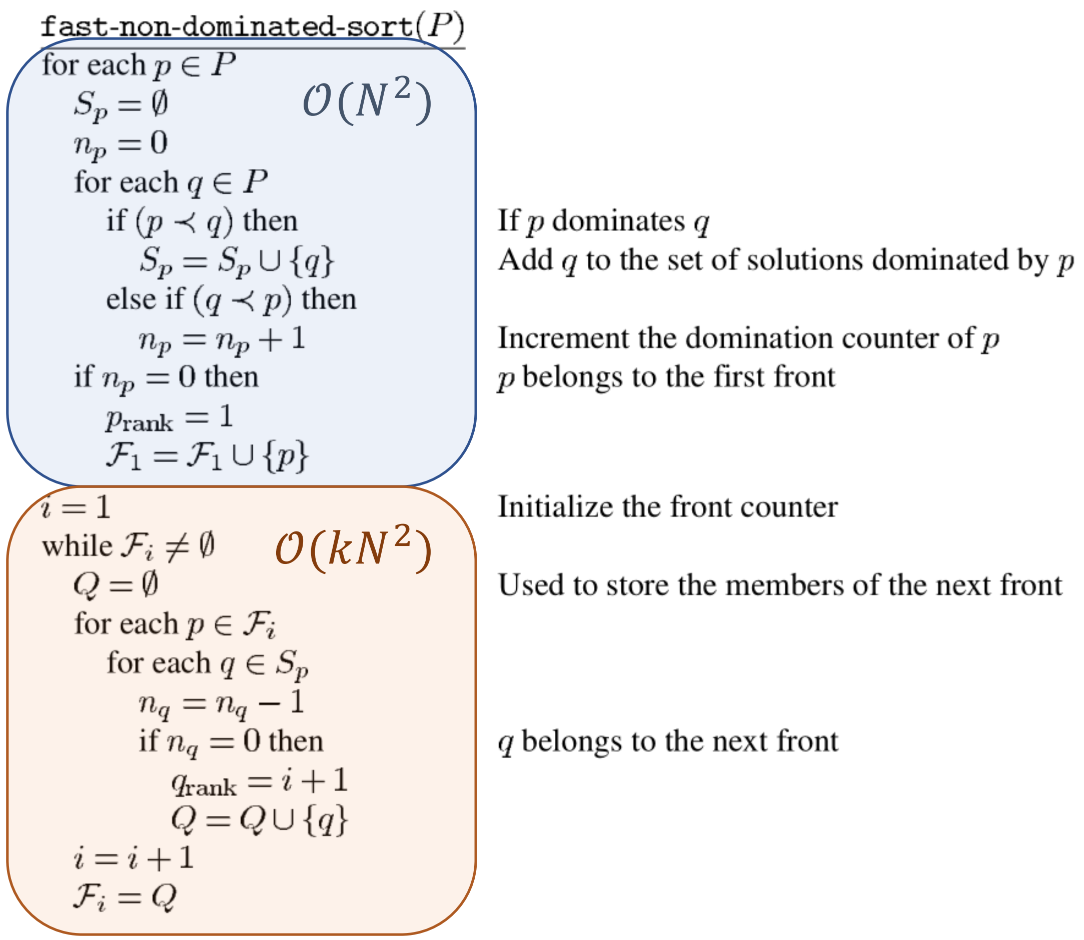
- Time complexity reduced to $\mathcal{O}(k \cdot N^2)$
- Spatial complexity increased to $\mathcal{O}(N^2)$
- [Deb *et al.*, IEEE Tran Evol Comp 2002]



```
fast-non-dominated-sort($P$)
for each $p \in P$                                          $\mathcal{O}(N^2)$
    $S_p = \emptyset$
    $n_p = 0$
    for each $q \in P$
        if $(p \prec q)$ then                               If $p$ dominates $q$
            $S_p = S_p \cup \{q\}$                          Add $q$ to the set of solutions dominated by $p$
        else if $(q \prec p)$ then
            $n_p = n_p + 1$                                 Increment the domination counter of $p$
    if $n_p = 0$ then                                       $p$ belongs to the first front
        $p_{\text{rank}} = 1$
        $\mathcal{F}_1 = \mathcal{F}_1 \cup \{p\}$
$i = 1$                                                     Initialize the front counter
while $\mathcal{F}_i \neq \emptyset$
    $Q = \emptyset$                                         Used to store the members of the next front
    for each $p \in \mathcal{F}_i$
        for each $q \in S_p$
            $n_q = n_q - 1$
            if $n_q = 0$ then                               $q$ belongs to the next front
                $q_{\text{rank}} = i + 1$
                $Q = Q \cup \{q\}$
    $i = i + 1$
    $\mathcal{F}_i = Q$
```

NSGA-II’s fast dominated sorting

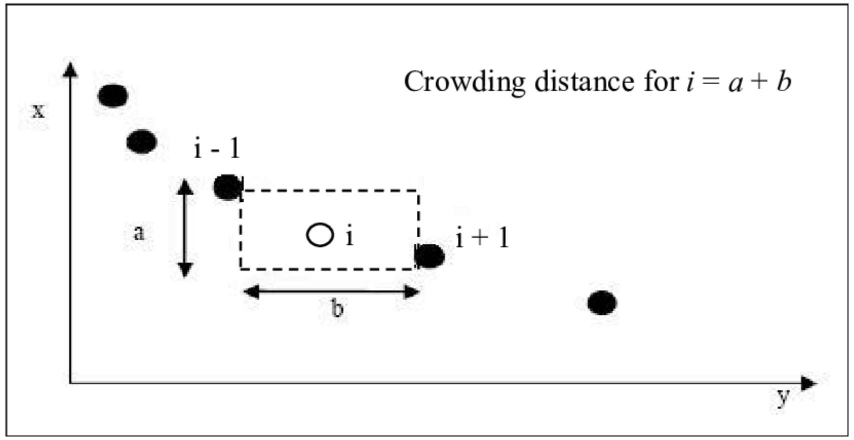
- Time complexity reduced to $\mathcal{O}(k \cdot N^2)$
- Spatial complexity increased to $\mathcal{O}(N^2)$
- [Deb *et al*., IEEE Tran Evol Comp 2002]

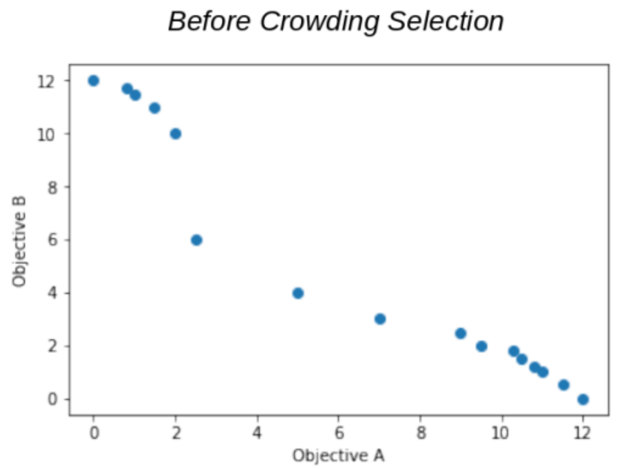


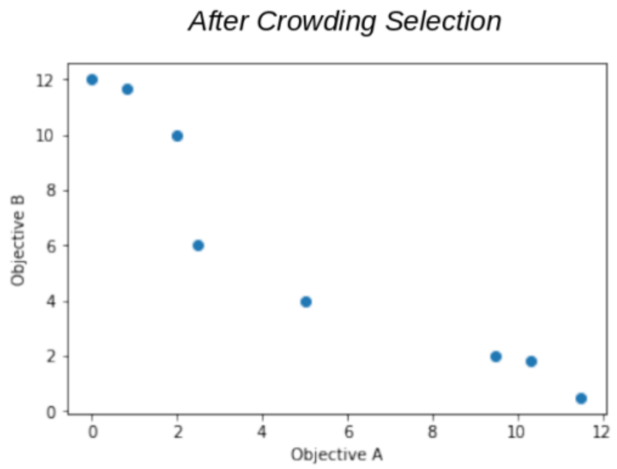
```
fast-non-dominated-sort($P$)
for each $p \in P$                                $\mathcal{O}(N^2)$
  $S_p = \emptyset$
  $n_p = 0$
  for each $q \in P$
    if $(p \prec q)$ then                         If $p$ dominates $q$
      $S_p = S_p \cup \{q\}$                      Add $q$ to the set of solutions dominated by $p$
    else if $(q \prec p)$ then
      $n_p = n_p + 1$                             Increment the domination counter of $p$
  if $n_p = 0$ then                               $p$ belongs to the first front
    $p_{\text{rank}} = 1$
    $\mathcal{F}_1 = \mathcal{F}_1 \cup \{p\}$
$i = 1$                                           Initialize the front counter
while $\mathcal{F}_i \neq \emptyset$              $\mathcal{O}(kN^2)$
  $Q = \emptyset$                                 Used to store the members of the next front
  for each $p \in \mathcal{F}_i$
    for each $q \in S_p$
      $n_q = n_q - 1$
      if $n_q = 0$ then                           $q$ belongs to the next front
        $q_{\text{rank}} = i + 1$
        $Q = Q \cup \{q\}$
  $i = i + 1$
  $\mathcal{F}_i = Q$
```

Crowding Distance (CD)

- To prevent the loss of diversity we need a «measure» of diversity
  - Diversity in **objective space**
  - **Crowding Distance** (CD) is one example of this measure
  - CD selection: less crowded solutions (belonging to the *same* front) have a **higher possibility of being selected** for the next generation

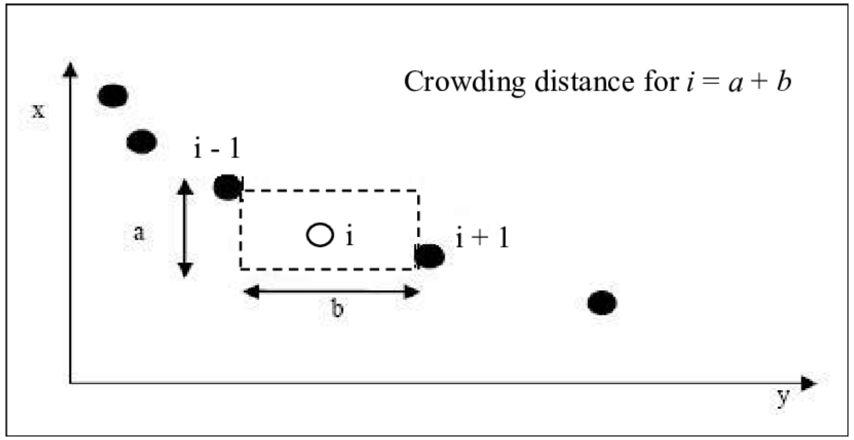


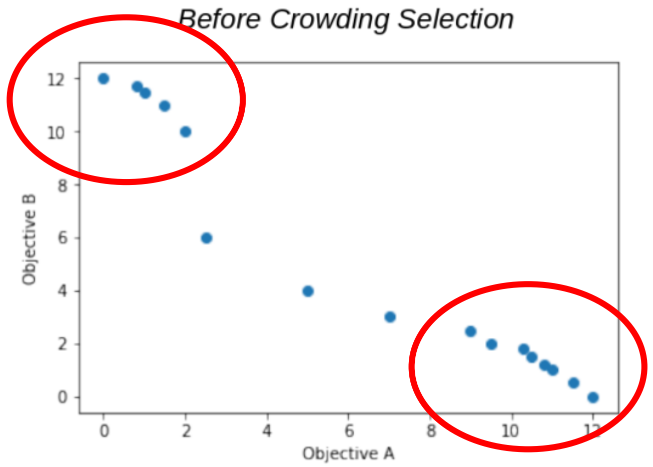


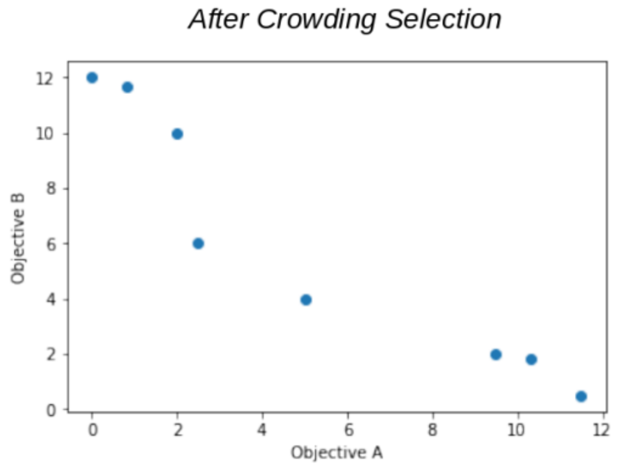


Crowding Distance (CD)

- To prevent the loss of diversity we need a «measure» of diversity
  - Diversity in **objective space**
  - **Crowding Distance** (CD) is one example of this measure
  - CD selection: less crowded solutions (belonging to the *same* front) have a **higher possibility of being selected** for the next generation



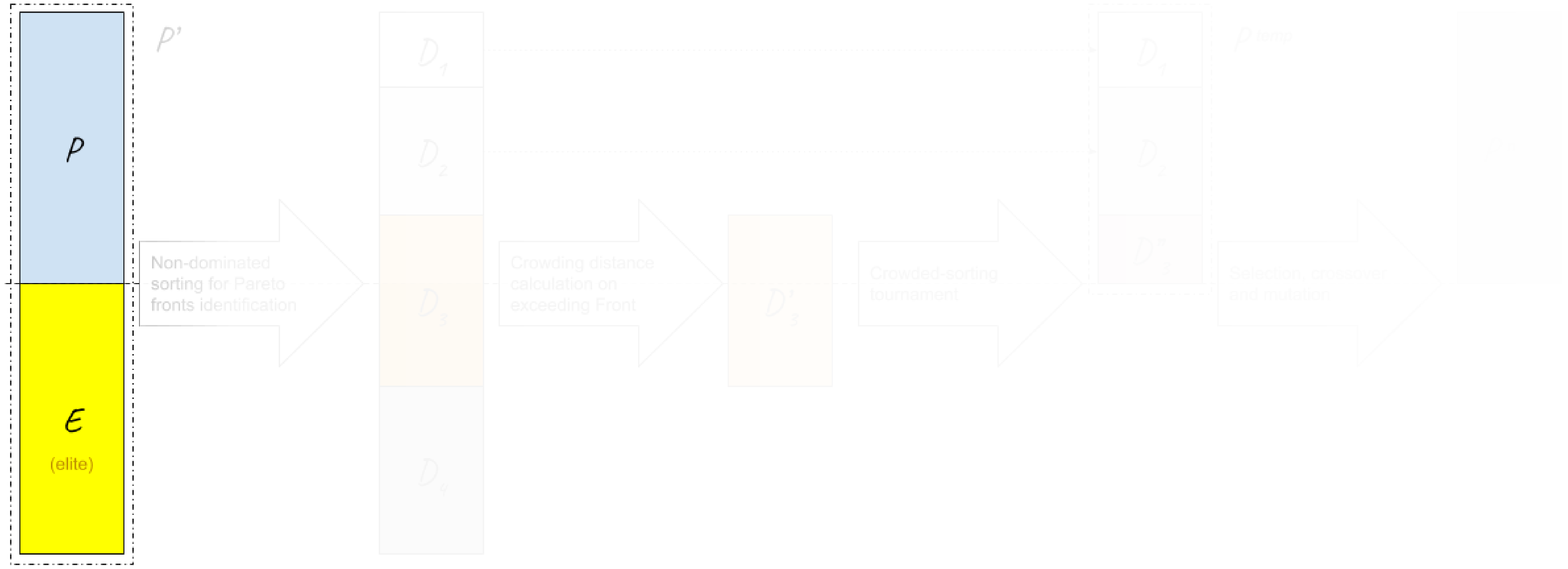




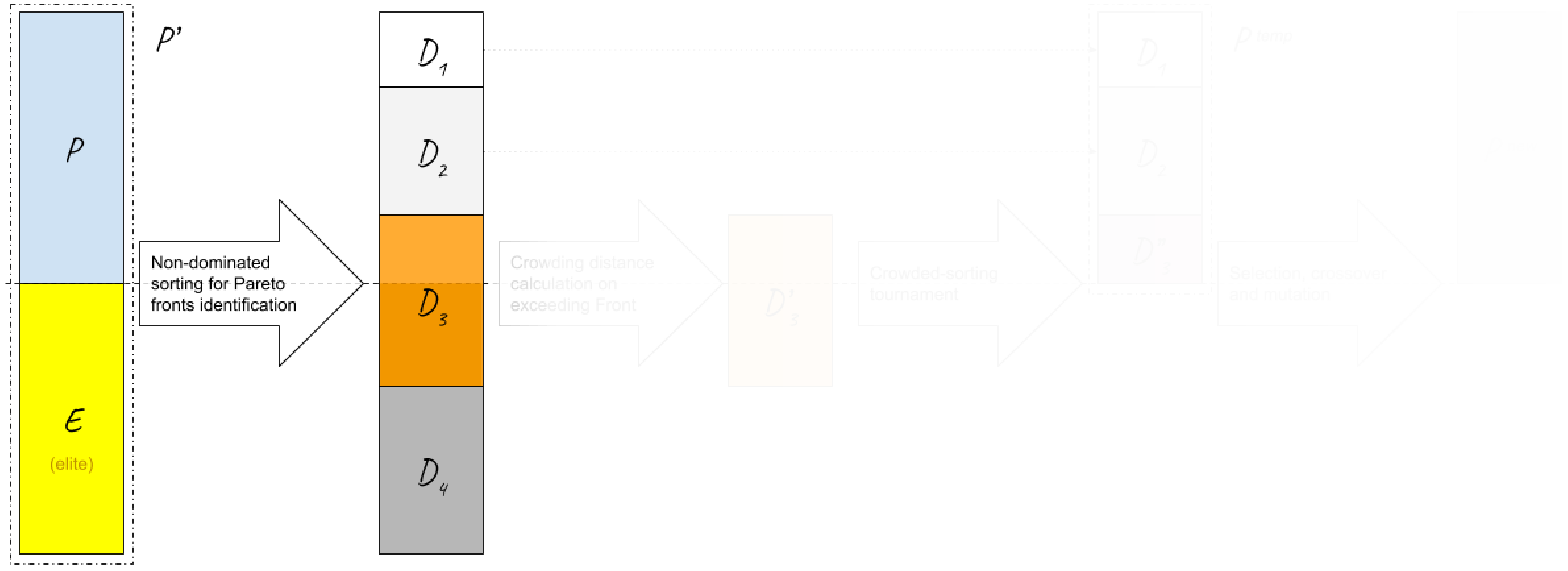
Crowded-comparison operator

- The crowded-comparison operator, denoted by $\prec_n$, guides the **selection process**
  - Designed to obtain a uniformly **spread out the Pareto front**
  - Assume that each individual of the population has a rank R_i and a crowding distance CD_i
  - Then $i \prec_n j$ if $R_i < R_j$
  - Otherwise, $i \prec_n j$ if $R_i = R_j$ and $CD_i > CD_j$
  - $\prec_n$ induces **a partial order** on the solutions
  - See [Deb *et al*., IEEE Tran Evol Comp 2002]

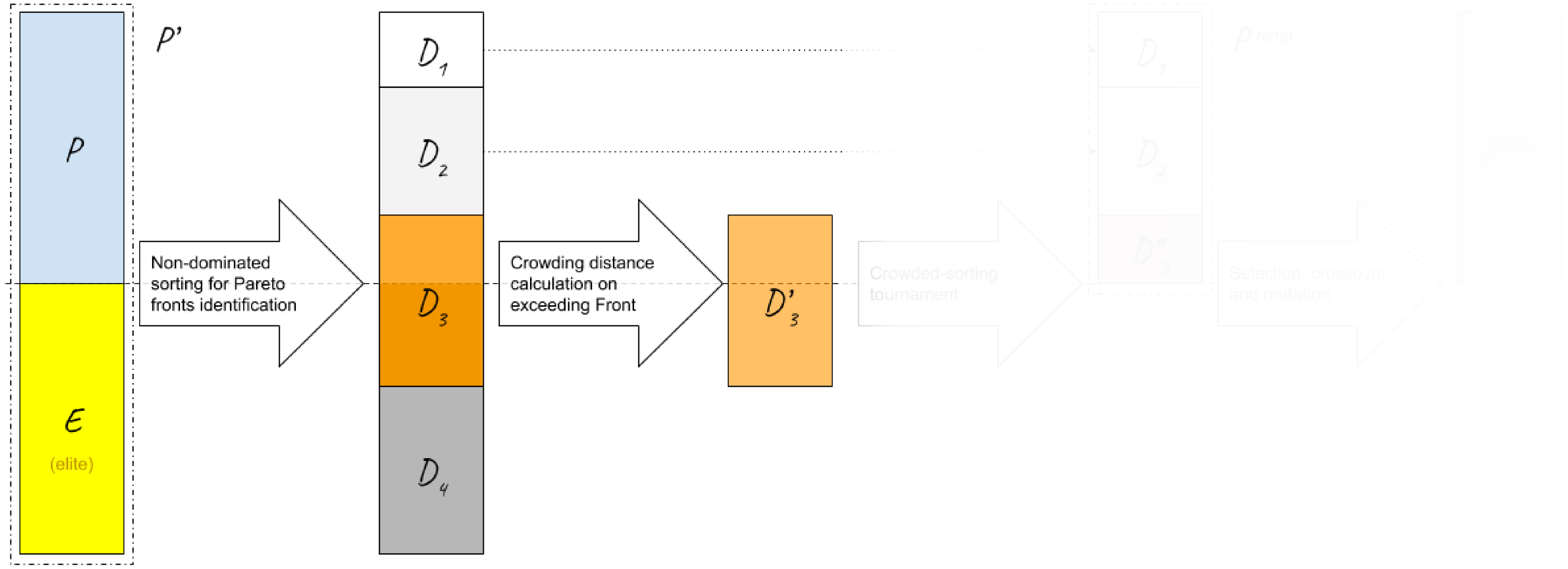
NSGA-II algorithm


P
E
(elite)
P'
Non-dominated sorting for Pareto fronts identification
D_1
D_2
D_3
D_4
Crowding distance calculation on exceeding Front
D'_3
Crowded-sorting tournament
Selection, crossover and mutation

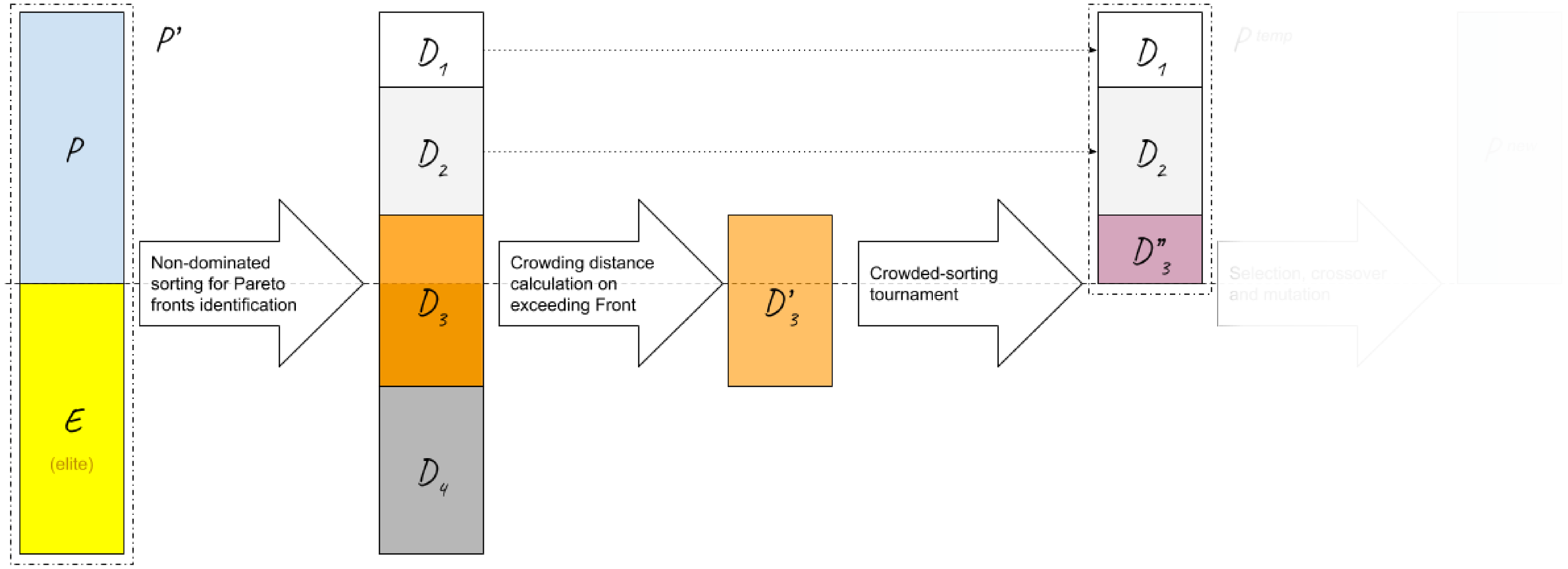
NSGA-II algorithm


P'
P
E
(elite)
Non-dominated sorting for Pareto fronts identification
D_1
D_2
D_3
D_4
Crowding distance calculation on exceeding Front
D'_3
Crowded-sorting tournament
D_1
D_2
D''_3
Selection, crossover and mutation

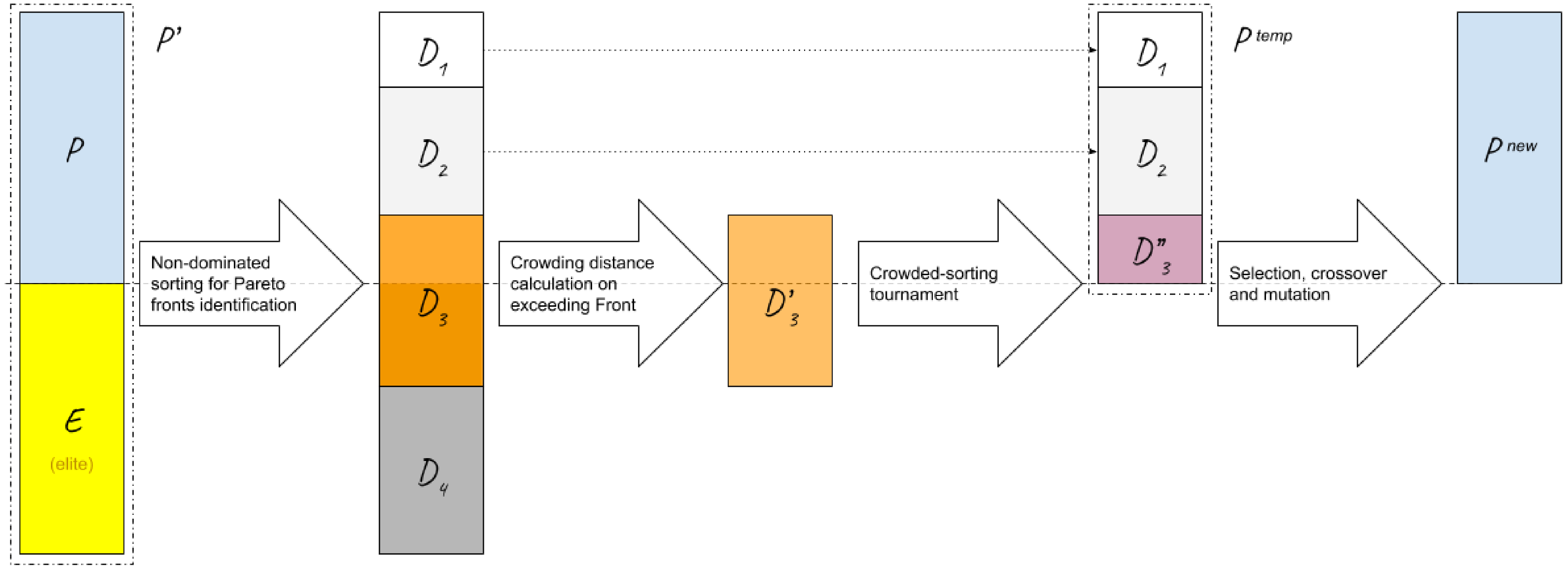
NSGA-II algorithm


P'
P
E
(elite)
Non-dominated sorting for Pareto fronts identification
D1
D2
D3
D4
Crowding distance calculation on exceeding Front
D'3
Crowded-sorting tournament
P temp
D1
D2
D''3
Selection, crossover and mutation

NSGA-II algorithm

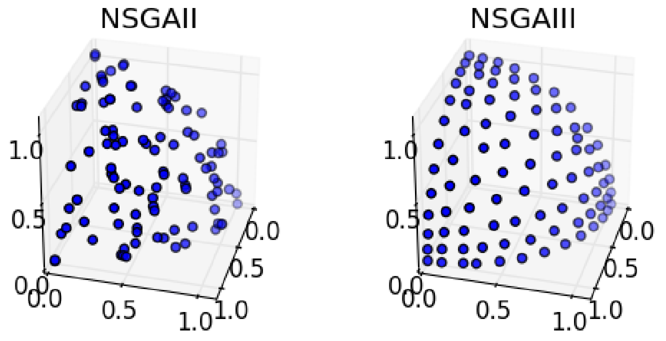

P'
P
E
(elite)
Non-dominated sorting for Pareto fronts identification
D_1
D_2
D_3
D_4
Crowding distance calculation on exceeding Front
D'_3
Crowded-sorting tournament
D_1
D_2
D''_3
P temp
Selection, crossover and mutation
P new

NSGA-II algorithm

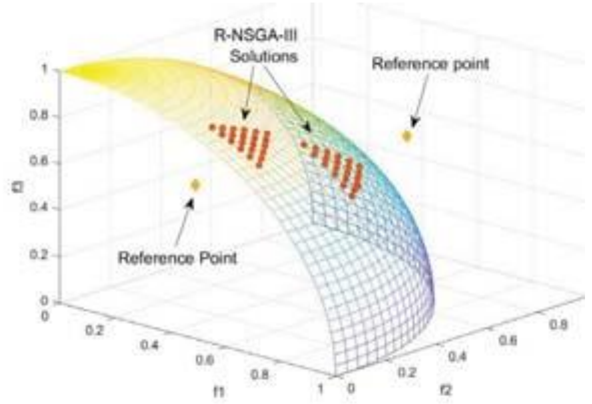

P'
P
E
(elite)
Non-dominated sorting for Pareto fronts identification
D1
D2
D3
D4
Crowding distance calculation on exceeding Front
D'3
Crowded-sorting tournament
P temp
D1
D2
D"3
Selection, crossover and mutation
P new

NSGA-III

- NSGA-III is a variant of NSGA-II for ***many* objectives optimization**
  - That is, three objectives or more
  - Exploits a set of **«reference points»** to preserve the diversity in the Pareto fronts
  - NSGA-III produces an **even distribution of solutions** across the objective space even when the number of objectives is large
  - [Deb and Jain, IEEE Tran Evol Comp 2014]



- R-NSGA-III: a variant of NSGA-III
  - Uses a new reference point generation based on **user-defined «aspiration points»** in the k-dimensional objective space
  - [Vesikar *et al*., IEEE Symposium Series on Computational Intelligence 2018]



MOPSO [1/3]

- **Multi-Objective Particle Swarm Optimization** (MOPSO)
    - Swarm Intelligence meta-heuristic for multi-objective optimization
    - Main idea: **change the particles' velocity update** wrt classic PSO to consider an attraction «towards the Pareto front», i.e., towards the **dominating particles** found so far
    - Secondary idea: maintain a **list of the hyper-cubes** that keep track of the dominating positions visited so far

MOPSO [2/3]

- In MOPSO, a global **repository of non-dominated particles $\boldsymbol{REP}$** is maintained
  - When a **new dominating solution** is added to $\boldsymbol{REP}$, the **dominated particles are removed**
  - The repository has a **limited size** Q
  - When the number of particles in the repository is greater than Q, the particles belonging to **most crowded** hyper-volumes are **removed** until the repository containes exactly Q particles

- MOPSO's velocity update formula becomes

$$\mathbf{v}_i(t+1) = w \cdot \mathbf{v}_i(t) + c_{soc} \cdot \mathbf{r}_1 \otimes \big(\boldsymbol{REP}_h - \mathbf{x}_i(t)\big) + c_{cog} \cdot \mathbf{r}_2 \otimes \big(\mathbf{p}_i - \mathbf{x}_i(t)\big)$$

  - $\boldsymbol{REP}_h$ is the h-th particle of the repository
  - The index h is determined using a **tournament selection** over the particles in the repository
  - **Fitness sharing** is used to reduce the fitness values of particles belonging to **crowded hyper-cubes**
  - The personal best $\mathbf{p}_i$ now corresponds to the best position visited so far by the i-th particle *wrt* all objectives, i.e., it is not updated when $\mathbf{p}_i \prec \mathbf{x}_i(t)$
  - [Coello-Coello and Sala, IEEE CEC 2002]

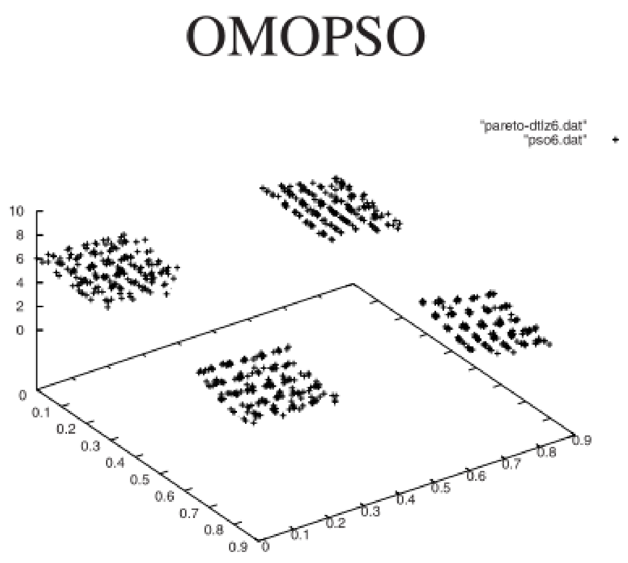
MOPSO [3/3]

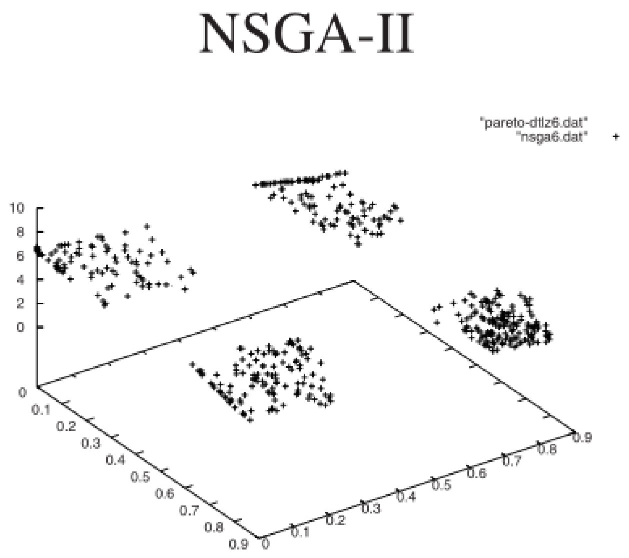
- MOPSO has several **hyper-parameters** to be tuned
  - The size of the repository
  - The size of the tournament
  - The size of hyper-cubes (i.e., the number of divisions)
  - The size of the swarm
  - The inertia factor
  - The social and cognitive factors
  - The maximum (and minimum) velocity
  - The number of iterations

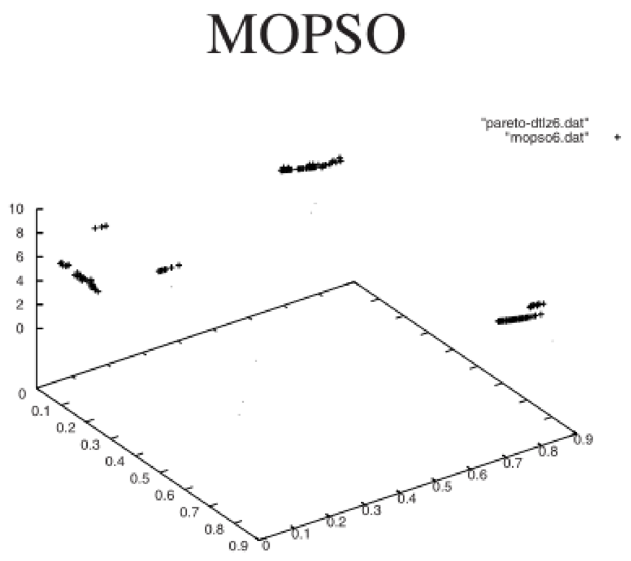
- Some of the aforementioned settings can be derived by **classic PSO**
  - Once again, the optimal settings are **problem-dependent**
  - **Adaptive methods** have been proposed, e.g., **pccsAMOPSO** [Hu and Yen, IEEE Tran Evol Comp 2013], **AMOPSO** [Han *et al*., IEEE Tran Cybern 2017], **AGMOPSO** [Han *et al*., IEEE Tran Cybern 2017]

OMOPSO

- Extends MOPSO by adding **mutation operators**
  - **The swarm is partitioned** into multiple subgroups: each subgroup uses a **different mutation operator**
  - Uniform operator promotes exploration, non-uniform operator promotes exploitation
  - Leverages ***ε*-dominance** to update the repository of dominating particles (additional hyper-parameters…)
  - [Sierra and Coello-Coello, EMO 2005]



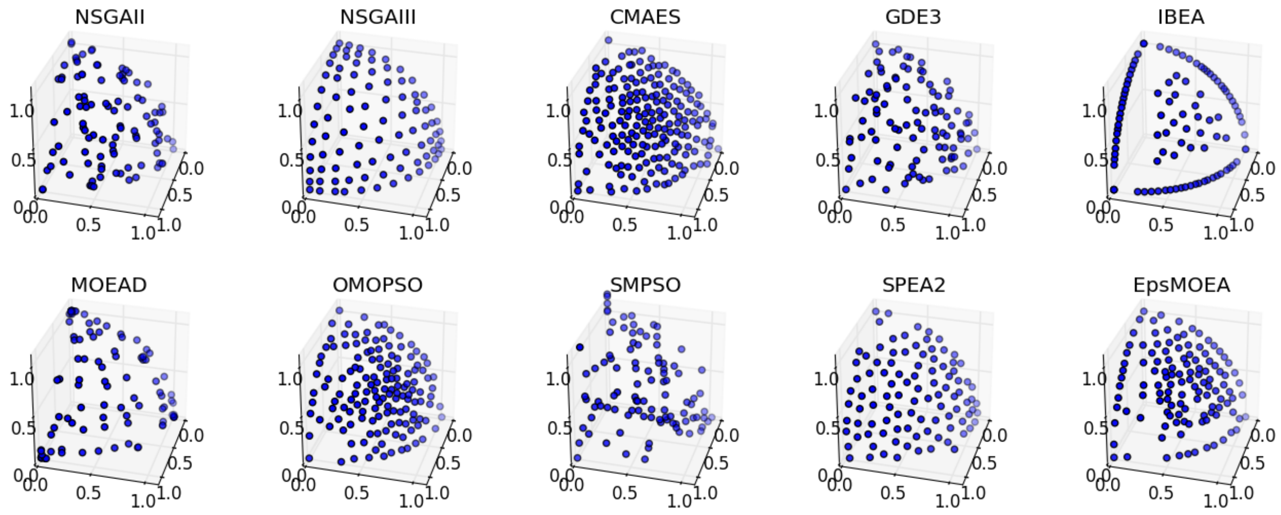




pip install platypus-opt

Out of the box MOO: Platypus

- **Platypus: MOO in python**
  - Supports a lot of widespread algorithms including NSGA-II, NSGA-III, OMOPSO, etc.
  - Integrates several MOO **benchmark functions**
  - + Easy to use – Documentation
  - Supports **automated testing** and visual comparison of performances (see Experimenter class)
  - **Supports parallel computation** (using multi-processing, process pools, etc.)
  - https://github.com/Project-Platypus/



Global Optimization and High Performance Computing

GPGPU computing & MPI

Outline

- Why
- Amdahl’s law
- Parallelization in evolutionary and swarm intelligence techniques
- High Performance Computing infrastructures
- GPGPU computing
- MPI

Motivation

- The optimization algorithms are generally tested on **benchmark functions**, i.e., the fitness evaluations are «simple» equations, **fast to calculate**
    - This is radically misleading, since **real world optimization problems** are a totally **different story**
- One example: biochemical parameter estimation
    - Fitness evaluation is based on biochemical reaction-based simulation (e.g., Gillespie’s SSA)
    - Running time of a <u>single</u> simulation in the case of a relatively small system: approx. 30 seconds
    - 100 particles $\times$ 1000 generations $\times$ 30 seconds = $3 \cdot 10^6$ seconds $\cong$ **5 weeks** (!)
- One possible solution: surrogate modeling (e.g., GP, Kriging, Gaussian mixtures, RDFs)
- Alternative option: **parallel computing**

Speeding up the calculations

- We denote by T_s the running time of (the most efficient implementation) of a serial heuristics
- We denote by $T_p(n)$ the running time of a parallel version of the same algorithm using n processors
- Then, the **speedup** provided by the parallel implementation is defined as $S_n = \frac{T_s}{T_p(n)}$
  - Please note that we can also assess a relative speedup by considering $S_n = \frac{T_p(1)}{T_p(n)}$

- Ideally, we aim at a **linear scaling of performances**, i.e., $S_n = n$
  - In that case, the higher the number of processors, the better the performances
  - This peak scaling is **difficult to achieve** in practice (e.g., memory access conflicts)
  - Still, there might be even the case of **superlinear speedup** (due to, e.g., cache, vector instructions) but it extremely rare
  - In particular, it depends of the characteristics of the algorithm

Amdahl’s law

- Amdahl’s law is a formula that gives the **theoretical peak speedup** of a task at fixed workload that can be expected using **multiple processors** (or cores):

$$S_n = \frac{1}{F_s + \frac{F_p}{P}}$$

- In the formula, $F_s = 1 - F_p$ is the «serial» fraction of code that **cannot be parallelized**
  - When $F_s = 1$ then $F_p = 0$ and $S_n = 1$ (i.e., no speedup) regardless the number of processors
  - When $F_p = 1$ then $F_s = 0$ and $S_n = \frac{1}{F_p/_P}$ which means that $\lim_{P \to \infty} S_n = \infty$
  - Hence, the speedup that can be achieved depends on both (1) the **portion of the code** that can be actually paralellized and (2) the number of **available processors**

Population based heuristics are intrinsically parallel algorithms

- Let’s give a look at the abstract code of a generic population-based meta-heuristic
  - N=individuals, D=dimensions

```
POP = Ø
for i=1 to N:
      POP.add(create_new_individual(D))
while not termination_criterion():
      for individual in POP:
            individual.evaluate_fitness()
      POP = update_population()
BEST = POP.detect_best()
return BEST
```

Population based heuristics are intrinsically parallel algorithms

- Let’s give a look at the abstract code of a generic population-based meta-heuristic
  - N=individuals, D=dimensions

```
POP = Ø
for i=1 to N:
      POP.add(create_new_individual(D))
while not termination_criterion():
      for individual in POP:
            individual.evaluate_fitness()
      POP = update_population()
BEST = POP.detect_best()
return BEST
```

All individuals are mutually independent

Distributing the creation of individuals

- If we have $P = N$ processors, we can easily parallelize the creation process by launching P threads that generate random vectors of length D
  - In the optimal case, the computational complexity reduces to $\mathcal{O}(D)$
  - This is **rarely** the case because, generally, for common CPUs $N \gg P$
  - This process could be offloaded to parallel co-processors like MICs, because $P \approx N$
  - There might be some issue with **non-reentrant RNGs**
  - The creation of the population is performed **just once** during the optimization, so it has a **limited impact to the overall execution time**

- *What else?*

Population based heuristics are intrinsically parallel algorithms

- Let’s give a look at the abstract code of a generic population-based meta-heuristic
  - N=individuals, D=dimensions

```
POP = Ø
for i=1 to N:
      POP.add(create_new_individual(D))
while not termination_criterion():
      for individual in POP:
            individual.evaluate_fitness()
      POP = update_population()
BEST = POP.detect_best()
return BEST
```

All fitness evaluations are mutually independent

Distributing the fitness evaluations

- This is a good idea, because the **fitness evaluations** are usually the most **time consuming** and **computationally challenging** part of the code
  - All fitness evaluations during a specific generation t are **mutually independent**
  - That is, the fitness evaluations at generation t must (or should? see, e.g., asynchronous PSO in [Mussi *et al*., GECCO 2011]) wait for the termination of the fitness evaluations at generation $t-1$
  - Once again, by having P processors, the running time due to fitness evaluations is approximately **reduced by a factor $\boldsymbol{P}$**

Population based heuristics are intrinsically parallel algorithms

- Let’s give a look at the abstract code of a generic population-based meta-heuristic
  - N=individuals, D=dimensions

```
POP = Ø
for i=1 to N:
      POP.add(create_new_individual(D))
while not termination_criterion():
      for individual in POP:
            individual.evaluate_fitness()
      POP = update_population()
BEST = POP.detect_best()
return BEST
```

The possibility to parallelize this step depents on the algorithm

Parallelizing the population update

- This step is **implemented differently** in various algorithms
- In **GAs** and similar the population update implies the selection of the individuals, the crossover between pairs and the mutation step: process is still **parallelizable**
- In **PSO** and similar the population update requires the **calculation of the new velocities** and the **positions update**
    - WARNING: the global best **g** used for velocities calculations might change during the process. Risk of **critical runs**. Safer to perform the update of **g** at the end of the velocities update
- In **DE** and similar the population update ultimately depends on the mutation scheme; however, it generally requires the **recombination of multiple candidates** and to perform some calculations with them: process is still parallelizable
    - WARNING: similarly to PSO, if we use *DE/best-to-current* there might be a risk of **critical runs**
    - WARNING: multiple individuals are used to perform a differential mutation and their position could change due to population update. Intermediate population, perhaps?

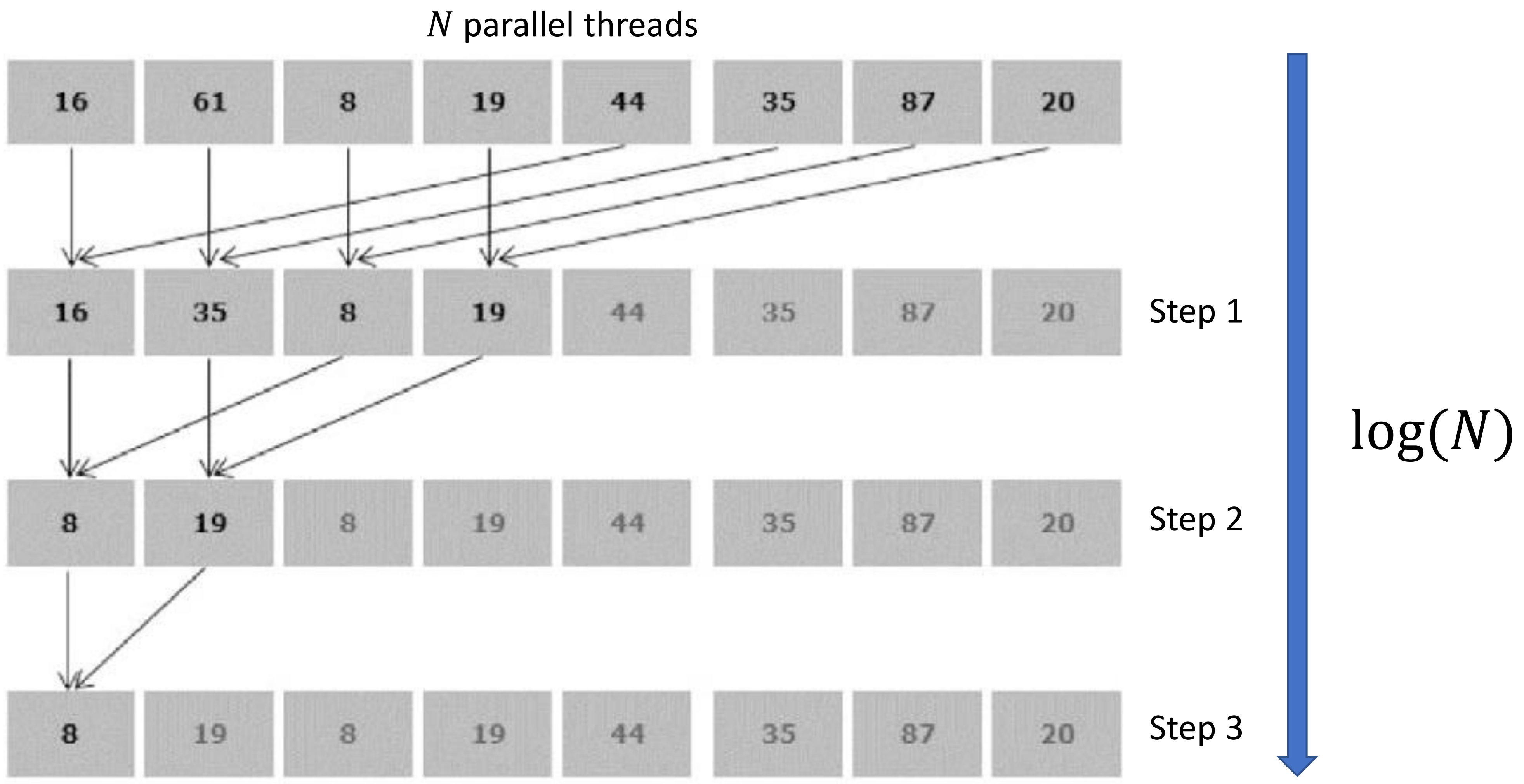
Population based heuristics are intrinsically parallel algorithms

- Let's give a look at the abstract code of a generic population-based meta-heuristic
  - N=individuals, D=dimensions

```
POP = Ø
for i=1 to N:
      POP.add(create_new_individual(D))
while not termination_criterion():
      for individual in POP:
            individual.evaluate_fitness()
      POP = update_population()
BEST = POP.detect_best()
return BEST
```

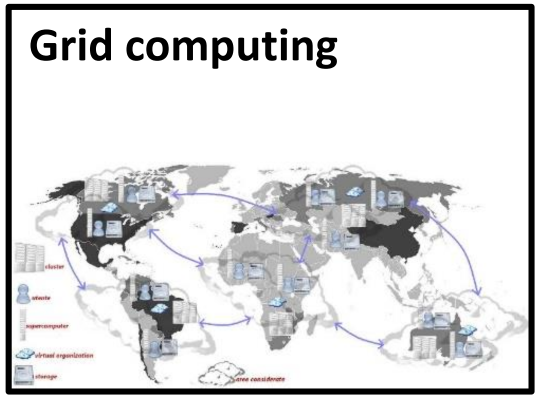
Parallel reduction can be used to find the minimum of a vector in $\mathcal{O}(\log(N))$

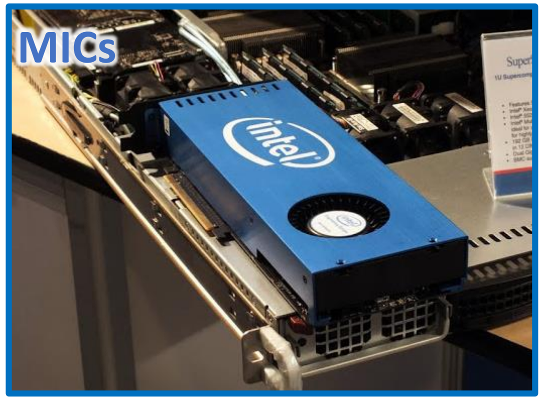
Example of parallel reduction to find the minumum

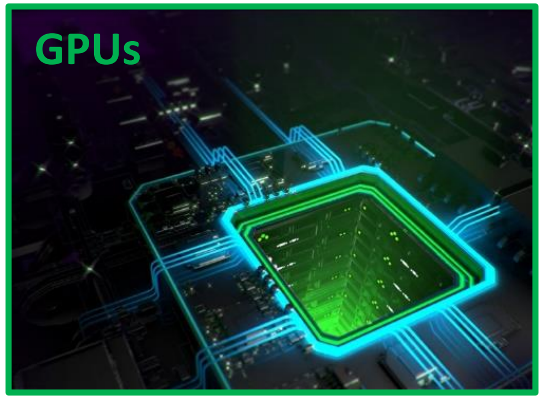


High Performance Computing

- There exist **several HPC architectures** that can be leveraged for **parallel computation**, each with different characteristics, peculiarities and drawbacks. In particular:



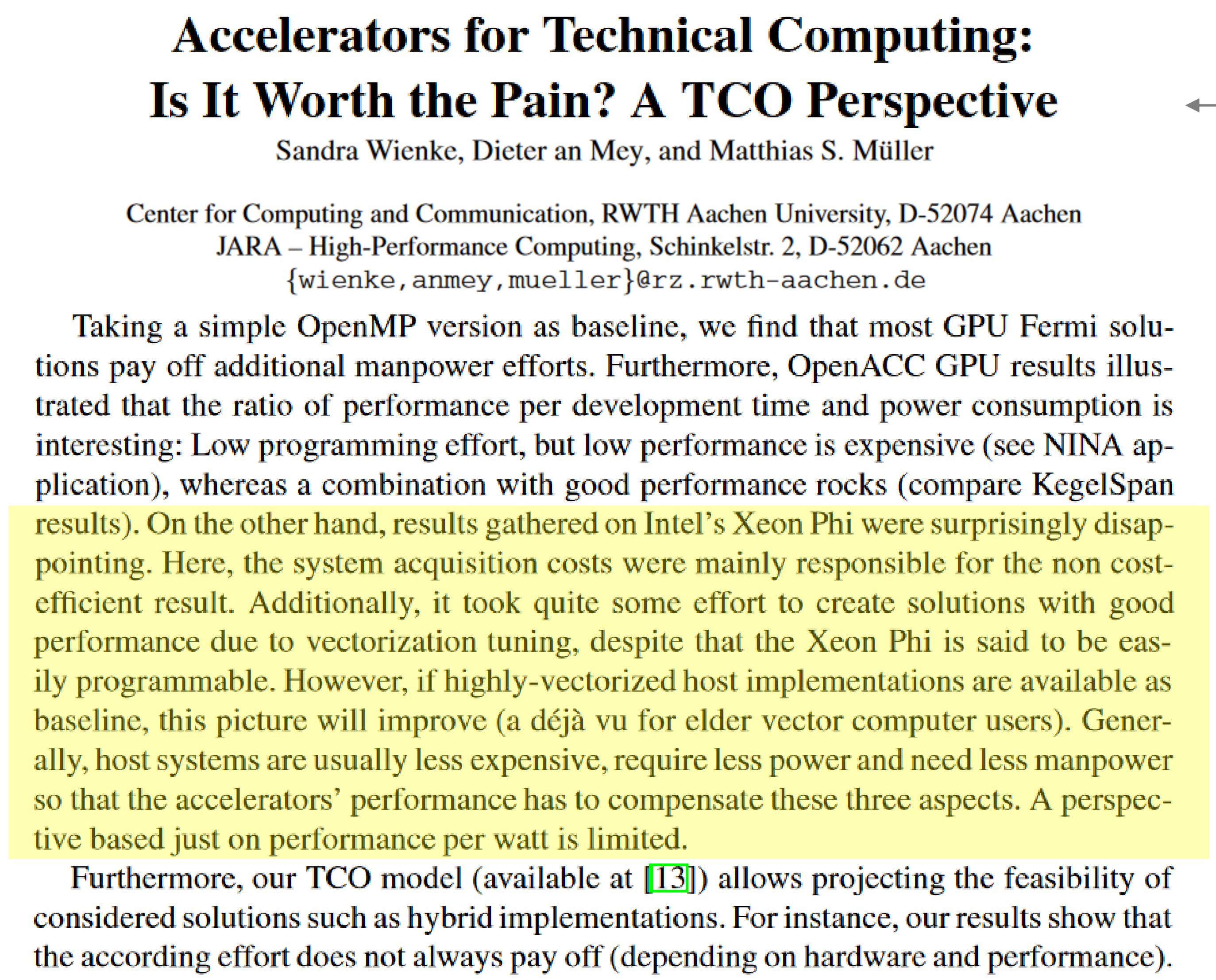




High-Performance Computing infrastructures

	PROS	CONS
Compute clusters	• Existing code is easy to port	• Large amount of energy is required (e.g., chinese Tianhe-1A: 4 MW) • Maintenance is required • Expensive infrastructure
Grid computing	• Existing code is easy to port • Costs are strongly reduced: no energy consumption nor maintenance	• Based on *volunteering*: no guarantees about the availability of remote computers • Some tasks could never be processed • Remote computers might not be completely trustworthy
Cloud computing	• Existing code is easy to port • Maintenance handled by the host • Backup handled by the host	• Resources owned by a private company • Issues of privacy (GDPR), piracy, espionage • International conflicts • Data *lock-in*, data transfer • Problems of continuity of service • Paid service
MICs	• Existing code is easy to port • Many cores (approx 70) • No maintenance (local execution) • TFLOP performances on a single PC	• Code needs to be rewritten/adapted to exploit vector instructions to reach peak performances • Not diffused (impossible on laptops)
GPUs	• Relatively cheap, highly diffused (pervasive), excellent W/GFLOPS ratio, thousands of cores • No maintenance (local execution) • TFLOP performances on a single PC	• Different architecture, programming model and ISA: porting may be difficult, code must be redesigned • «Relaxed» SIMD paradigm

MICs criticalities



Accelerators for Technical Computing:
Is It Worth the Pain? A TCO Perspective

Sandra Wienke, Dieter an Mey, and Matthias S. Müller

Center for Computing and Communication, RWTH Aachen University, D-52074 Aachen
JARA – High-Performance Computing, Schinkelstr. 2, D-52062 Aachen
`{wienke,anmey,mueller}@rz.rwth-aachen.de`

Taking a simple OpenMP version as baseline, we find that most GPU Fermi solutions pay off additional manpower efforts. Furthermore, OpenACC GPU results illustrated that the ratio of performance per development time and power consumption is interesting: Low programming effort, but low performance is expensive (see NINA application), whereas a combination with good performance rocks (compare KegelSpan results). On the other hand, results gathered on Intel's Xeon Phi were surprisingly disappointing. Here, the system acquisition costs were mainly responsible for the non cost-efficient result. Additionally, it took quite some effort to create solutions with good performance due to vectorization tuning, despite that the Xeon Phi is said to be easily programmable. However, if highly-vectorized host implementations are available as baseline, this picture will improve (a déjà vu for elder vector computer users). Generally, host systems are usually less expensive, require less power and need less manpower so that the accelerators' performance has to compensate these three aspects. A perspective based just on performance per watt is limited.

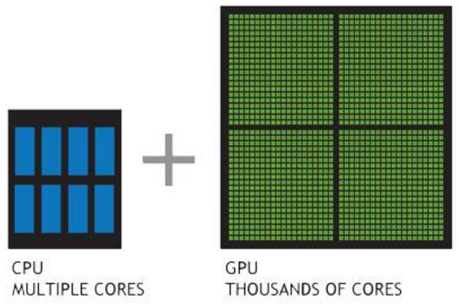
Furthermore, our TCO model (available at [13]) allows projecting the feasibility of considered solutions such as hybrid implementations. For instance, our results show that the according effort does not always pay off (depending on hardware and performance).

TCO: Total Cost of Ownership
Not just watt consumed, but also maintainance and programming effort

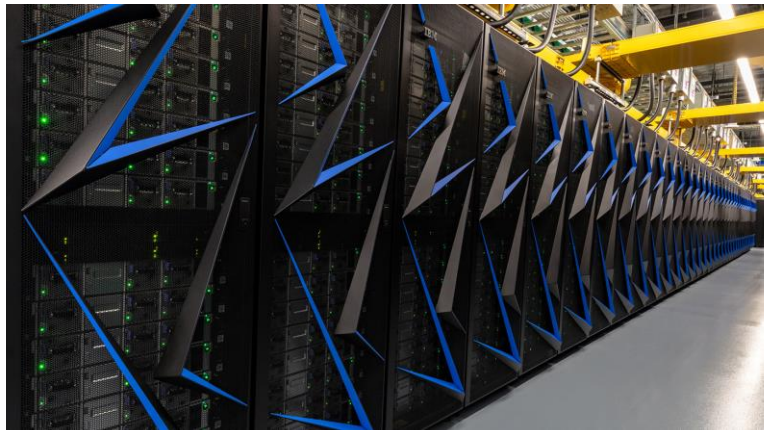
General-Purpose GPU computing

- GPGPU computing: the idea of leveraging the **massive parallelism of modern video-cards** for general-purpose computation
  - [Nickolls et al., ACM Queue 2008]

- GPUs are **multi-core parallel co-processors**
  - One GPU = **thousands of cores**
  - Execution is asynchronous wrt to CPU: the two architectures can be simultaneously exploited



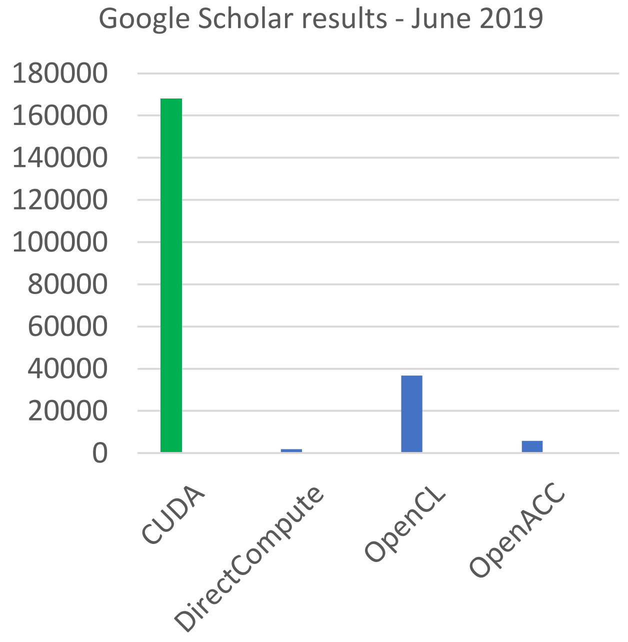
- Recent GPUs achieve **teraflop performances**
  - Nvidia RTX 2080 Ti FE: **4352 cores**, **14.2 TFLOPS**
  - World's most powerful **supercomputers** are GPU-based: **ORNL's Summit** (200 petaflops) and LLNL's Sierra (125 petaflops)
  - https://www.top500.org/lists/2018/11/



GPGPU computing solutions

- There exist **multiple solutions**/libraries for **GPGPU computing**
  - OpenCL
  - Microsoft DirectCompute
  - OpenACC
  - **Nvidia CUDA**

- Nvidia CUDA is, by far, the **most widespread technology** in the scientific literature
  - Mature architecture, many **scientific libraries** available (e.g., DL)
  - Support for all operating systems (e.g., MS Windows, Apple OS X, Linux)
  - **It works only on Nvidia GPUs**
  - Hundreds of GPU-powered tools for Bioinformatics, Systems Biology, and Computational Biology [Nobile *et al.*, Brief Bioinform 2016]
  - CUDA will be considered in the rest of the lecture



Compute Unified Device Architecture (CUDA)

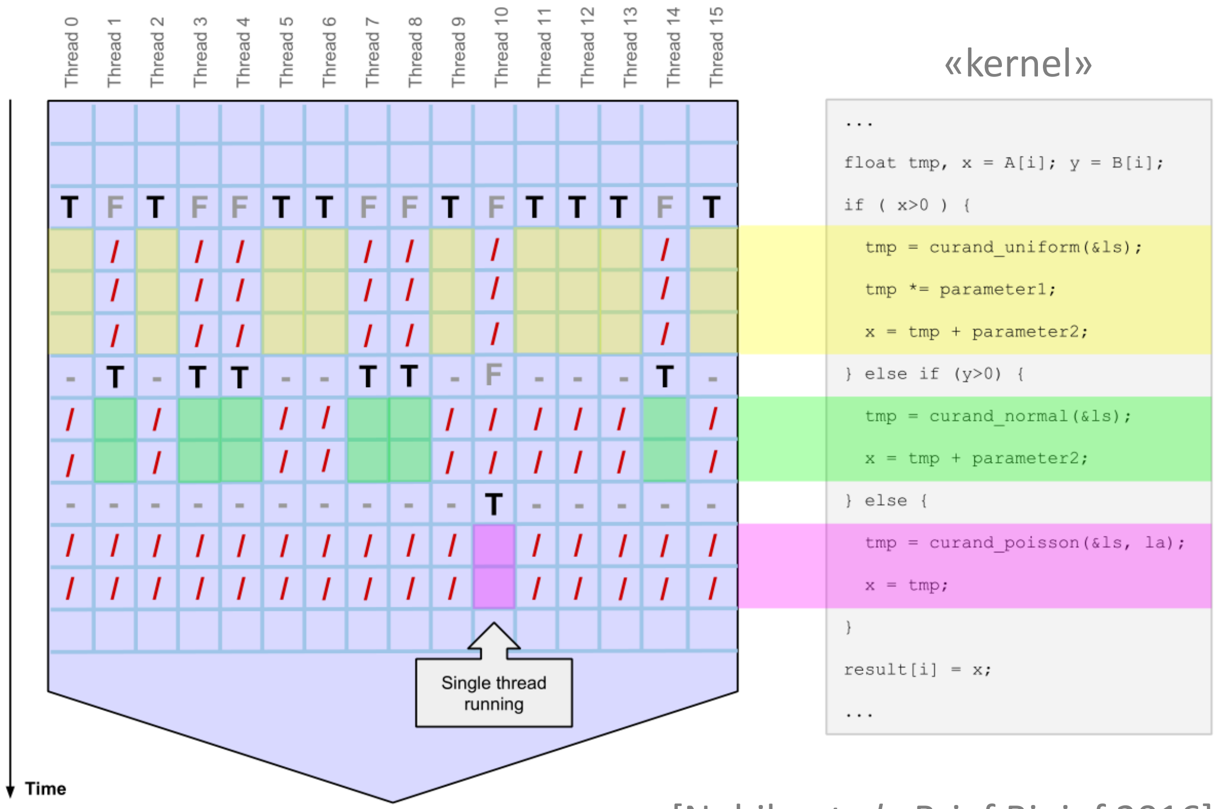
- CUDA is based on an **extension of the C/C++ language**
  - However, there are **bindings** for any programming language (e.g., Python, Java)
  - **Native support for FORTRAN**

- Existing algorithms **cannot be straightforwardly ported** to CUDA
  - CUDA relies on a **special execution paradigm** named «same-instruction, multiple threads» (SIMT), which is very different wrt classic multi- or many-cores CPUs
  - CUDA introduces a **special execution hierarchy**
  - CUDA relies on a **peculiar memory hierarchy**
  - All these characteristics **must be leveraged** and **optimized** to obtain good performances, mainly because **GPUs' clock frequency** is smaller than CPU (Nvidia GeForce RTX 2080 FE runs at 1800 MHz)
  - **The PCI-express bus is a relevant bottleneck** (avoid memory transfers as much as possible)

CUDA execution paradigm

- SIMT = a «relaxed» **same-instruction multiple-data** (SIMD) paradigm
  - Multiple cores → multiple threads
  - All cores are supposed to perform the **same computation** (e.g., simulation algorithm) working on **different data** (e.g., model parameterizations)
  - However, cores can (temporarily) take **divergent paths**
  - Divergence implies serialization → **reduced performances**

- Take home message: **stick to SIMD**

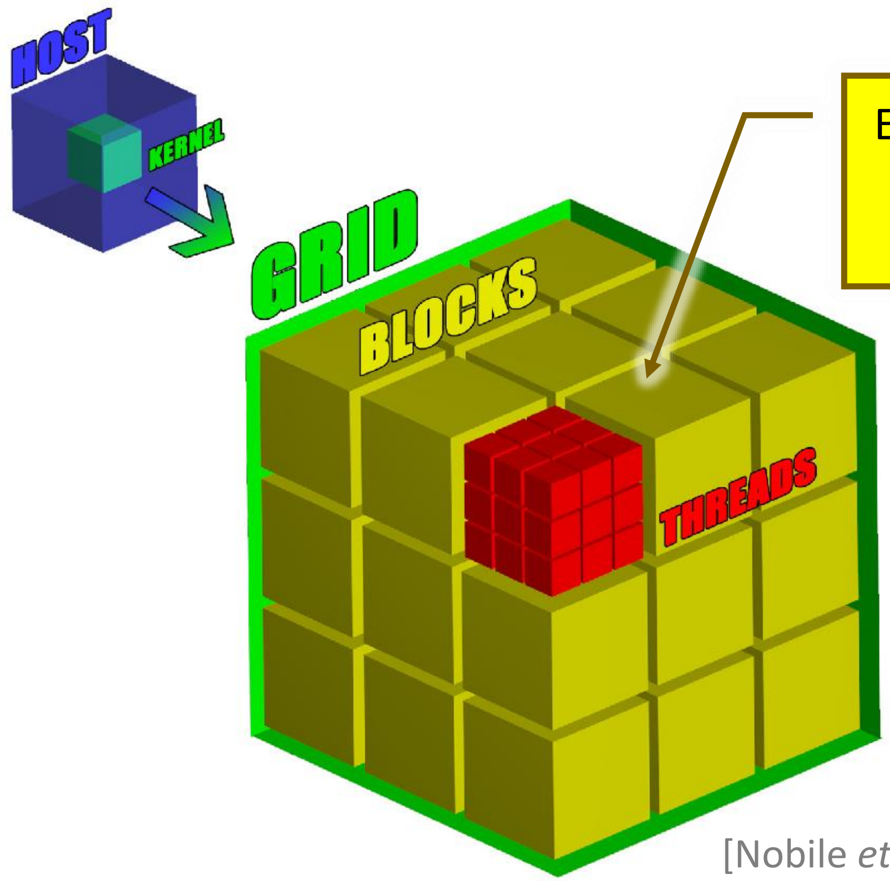


```
...
float tmp, x = A[i]; y = B[i];
if ( x>0 ) {
  tmp = curand_uniform(&ls);
  tmp *= parameter1;
  x = tmp + parameter2;
} else if (y>0) {
  tmp = curand_normal(&ls);
  x = tmp + parameter2;
} else {
  tmp = curand_poisson(&ls, la);
  x = tmp;
}
result[i] = x;
...
```

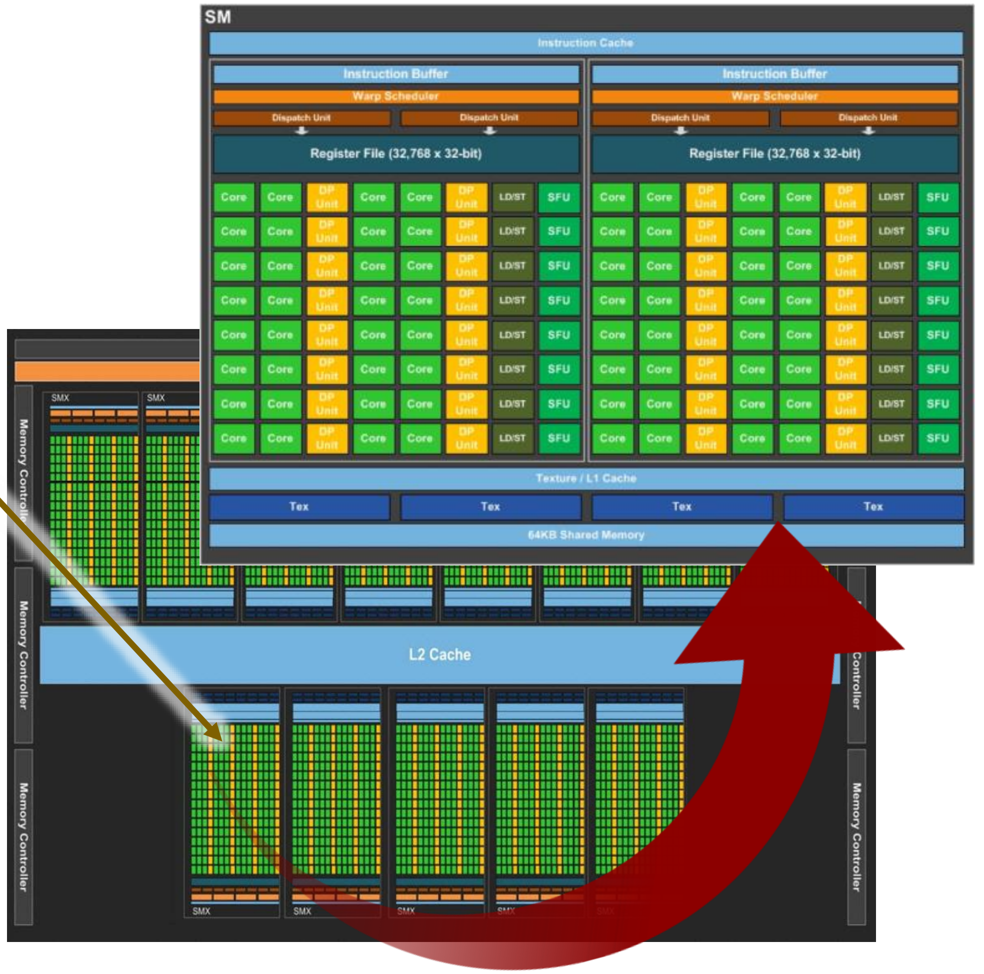
[Nobile *et al.*, Brief Bioinf 2016]

CUDA execution hierarchy

- GPU (*device*) is **architecturally very different** wrt CPU (*host*)
    - CUDA introduces the concept of **execution hierarchy**



Each block is assigned to an available multi-core streaming multiprocessor



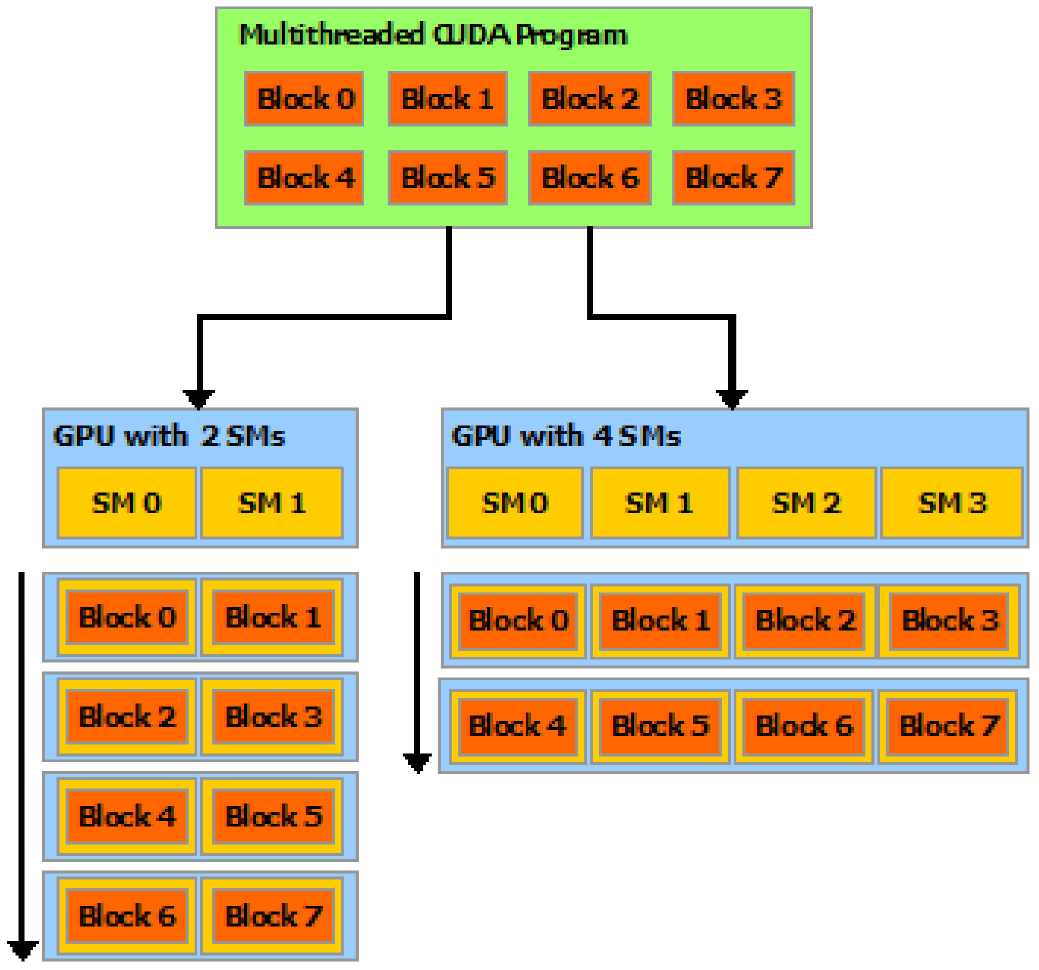
[Nobile *et al.*, Brief Bioinf 2016]

Scaling of performances

- All GPUs are different
    - In particular, different GPUs contain a different number of **Streaming Multiprocessors (SM)**
    - CUDA schedules each block to an **available SM**
    - The number of parallel running threads is roughly **proportional to the number of SMs**

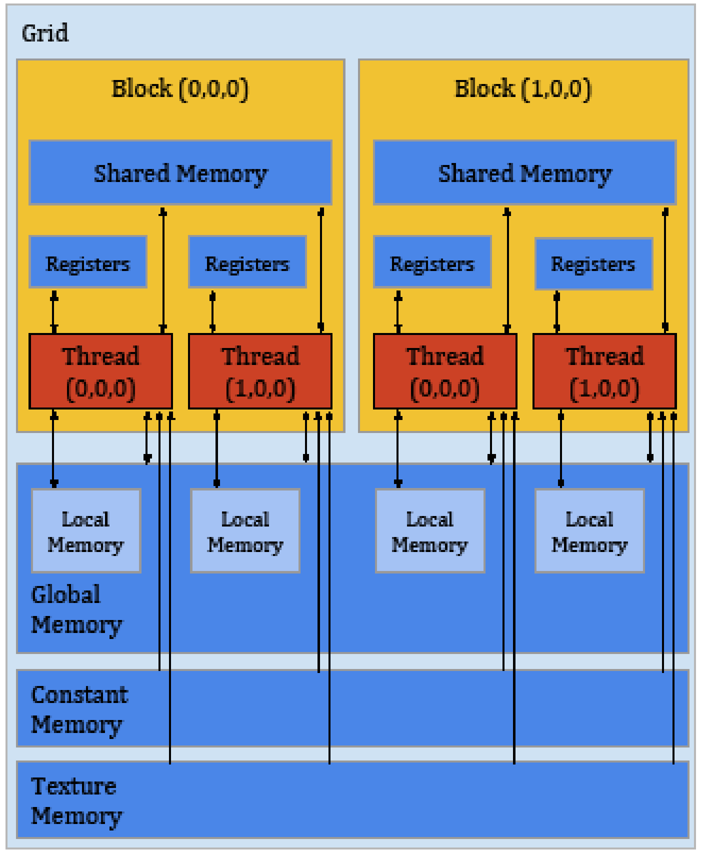
- **Transparent scaling of performances** on different GPUs

- **Automatic scaling on the latest GPUs**



CUDA memory hierarchy

- CUDA also introduces the **memory hierarchy**
  - Different memories with different «visibility», «speed» (i.e., access latencies) and «life-time»
- **Registers**: thread-level variables, extremely fast
- **Shared memory**: memory areas used for intra-block communication, very fast but very small (few KBs)
- **Global memory**: grid-level memory area, used as interface with the host, large but slow ($\approx 100 \times$ wrt shared memory)
- **Local memory**: chunks of global memory with thread-level visibility
- **Constant memory**: cached and read-only global memory
- **Texture memory**: cached global memory with additional circuitry for data interpolation
- **…and more!**



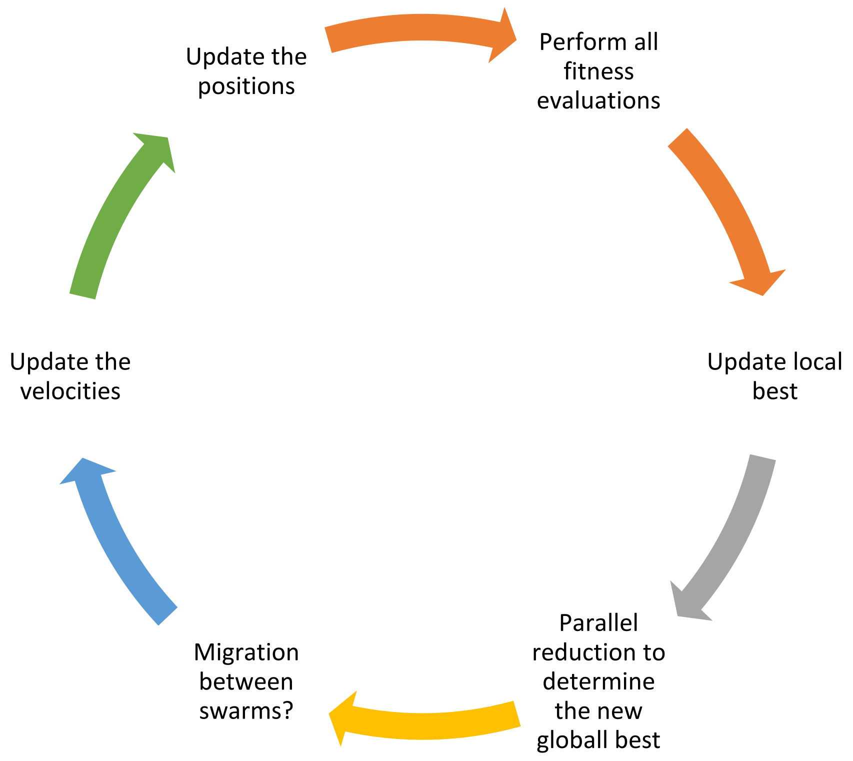
[Nobile *et al.*, Brief Bioinf 2016]

A straightforward example

- Multi-swarm PSO on GPUs
  - Problem to be solved: **parameter estimation of stochastic cellular systems**
  - **Multi-island paradigm** (island=swarm): separate swarms using **different experimental** data
  - **Periodic migrations of best individuals** between swarms with random topology to «distribute» potentially good parameterizations
  - Computational effort: 10 swarms × 100 particles × 1000 generations = 10^6 stochastic simulations. Assuming 30 s per simulation a whole run corresponds to approximately **one year**: it is **not feasible**
  - However, we can **synchronize the swarms** and **parallelize the operations: particles generation, velocity updates, positions update, fitness evaluations**
  - By exploiting $\mathbf{10 \times 100}$ **processors** the running time would be reduced **by three orders of magnitude**
  - We could build a cluster, offload to the GRID, buy some cloud computing time… or just use a **GPU**
  - [Nobile *et al*., EvoBIO 2012]

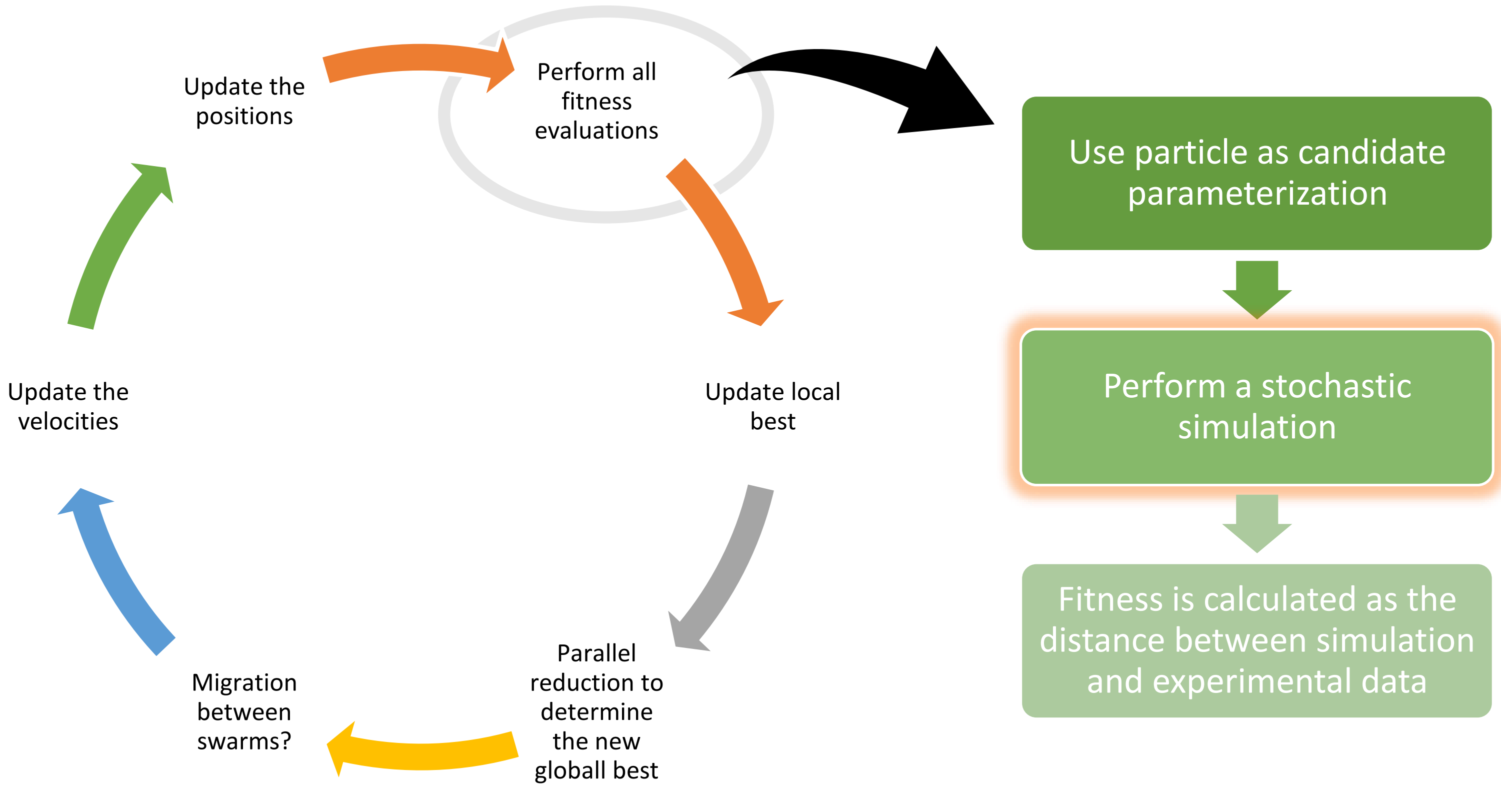
cuPEPSO's original implementation

- cuPEPSO: CUDA-powered Parameter Estimation with PSO
  - Everything is executed **on the device**, i.e., the CPU is almost completely idle with the **exception of memory transfers** due to initial **data load** and the **copy of results at the end**
  - **Monolithic implementation**: each particle is assigned to a specific thread; its fitness is evaluated on the GPU, i.e., the stochastic biochemical simulator is «contained» in the overall implementation


Update the positions
Perform all fitness evaluations
Update local best
Parallel reduction to determine the new globall best
Migration between swarms?
Update the velocities

cuPEPSO’s original implementation

- cuPEPSO: CUDA-powered Parameter Estimation with PSO
  - Everything is executed **on the device**, i.e., the CPU is almost completely idle with the **exception of memory transfers** due to initial **data load** and the **copy of results at the end**
  - **Monolithic implementation**: each particle is assigned to a specific thread; its fitness is evaluated on the GPU, i.e., the stochastic biochemical simulator is «contained» in the overall implementation


Update the positions
Perform all fitness evaluations
Update local best
Parallel reduction to determine the new globall best
Migration between swarms?
Update the velocities
Use particle as candidate parameterization
Perform a stochastic simulation
Fitness is calculated as the distance between simulation and experimental data

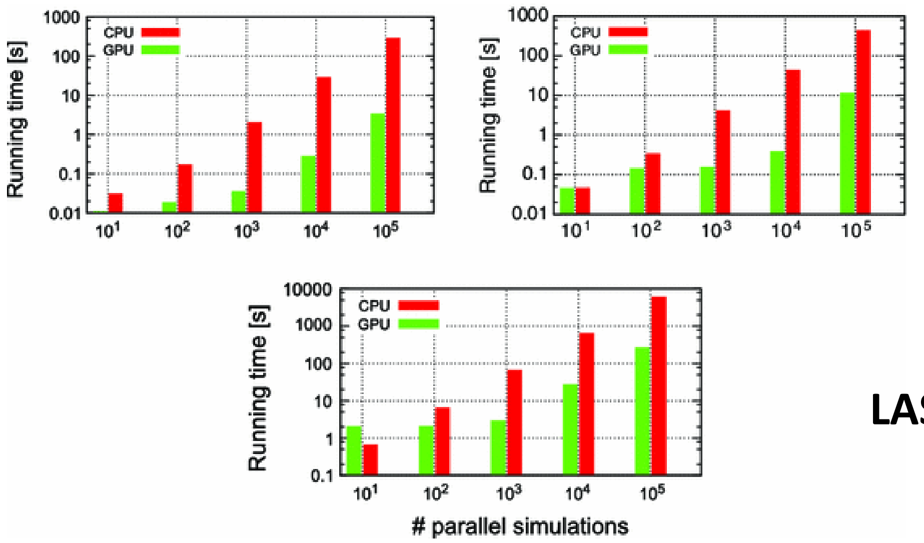
The cuPEPSO lesson

- We quickly realized that cuPEPSO's architecture was actually bad
  - Implementing the multi-swarm PSO on the GPU was not worth the effort: the algorithm has a **negligible impact** to the overall running time (99% of the time is due to fitness evaluations)
  - Better to **decouple** the fitness evaluation (i.e., simulation method) from the optimization algorithm
  - Maintaining the simulators and the optimization methods in the same CUDA/C++ project quickly becomes overly complicated
  - However, only a single CUDA context can be created simultaneously

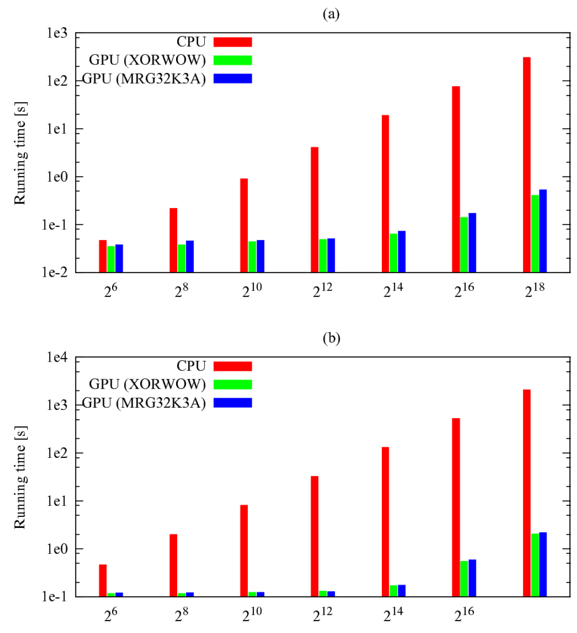
- Hence, we created **GPU powered simulators** to be embedded in our optimization methods
  - cuTauLeaping [Nobile *et al*., PLOS ONE 2014]
  - cupSODA [Nobile *et al*., J Supercomp 2014]
  - PySB/cupSODA [Harris *et al*., Bioinformatics 2017]
  - LASSIE [Tangherloni *et al*., BMC Bioinformatics 2017]
  - ginSODA [Nobile *et al*., J Supercomp 2019 (in press)]
  - LASSIE 2 (aka FiCoS) [Totis *et al*., LNBI 2019 (in press)]

Gain in performances

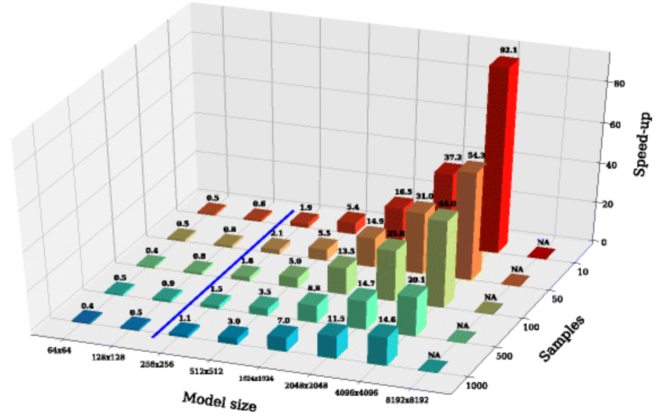
cupSODA: speedup up to 100x



cuTauLeaping: speedup up to 1000x



LASSIE: speedup up to 100x



GPU == panacea?

- Unfortunately not. Some tasks just cannot be implemented on/offloaded to a GPU
  - For instance, some tools cannot be ported for technical limitations, or some libraries are missing on the CPU, the DRAM is insufficient, etc.
  - In this case we can use alternative, for instance **MPI**

- MPI: **Message Passing Interface**
  - Standardized and portable **message-passing**
  - Provides communication topology, synchronization primitives (e.g., barriers), asynchronous communication, scattering, buffering…
  - Designed to work on a **variety** of parallel computing architectures
  - Supports C, C++, Fortran, Python, R, MATLAB
  - Goals of MPI: **high performance**, **portability**, **scalability**
  - *De facto* standard for **communication between processes**

Ranks

- When a MPI program is launched, a number of processes is requested
    - All of them run the **same source code** (obviously taking different branches on different processors)
    - The number of a specific running process is called **RANK**
    - The total number of running processes (i.e., the highest rank) is called **SIZE**
    - This information can (must) be used to determine «who is who» at run-time, in order to guide the execution and the communication

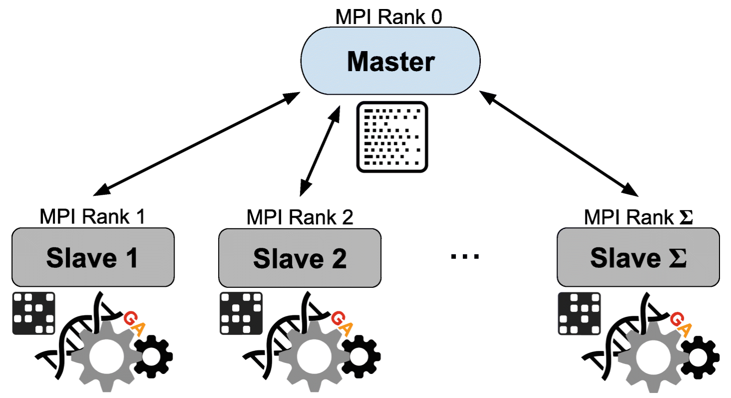
- A simple example using MPI4PY

```
from mpi4py import MPI

comm = MPI.COMM_WORLD                     # all processes belong to the same comm context
rank = comm.Get_rank()

if rank == 0:                             # This branch is taken by process w/ rank 0
    data = {'a': 7, 'b': 3.14}            # Any type of object can be set in a message
    comm.send(data, dest=1, tag=11)
elif rank == 1:                           # This branch is taken by process w/ rank 1
    data = comm.recv(source=0, tag=11)
```

One simple MPI architecture: master-slave

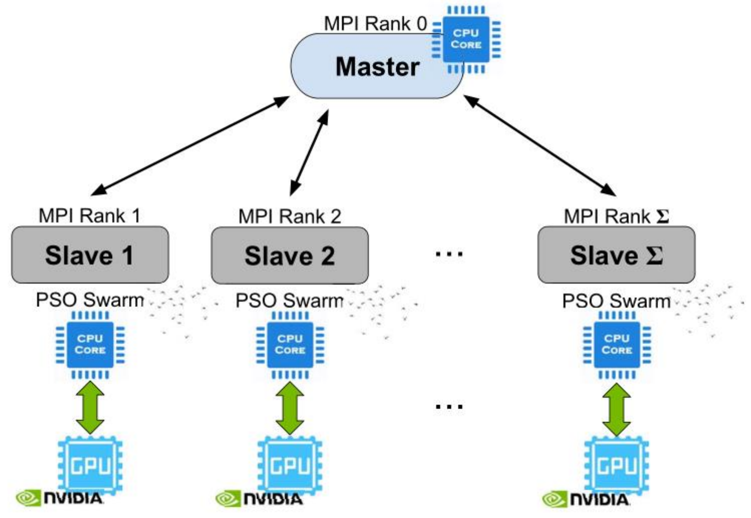
- The **Master-Slave** architecture is the most straightforward way to exploit MPI
  - One process (the «Master») controls the execution of the program and **sends messages** to…
  - …the rest of the processes (the «Slaves») which exploit such messages (e.g., candidate solutions) to execute some **useful operations** (e.g., fitness evaluations) on **separate processors** (e.g., cores/nodes)
  - When it's done, a **Slave sends a message to the master** (e.g., the fitness value) and waits further instructions

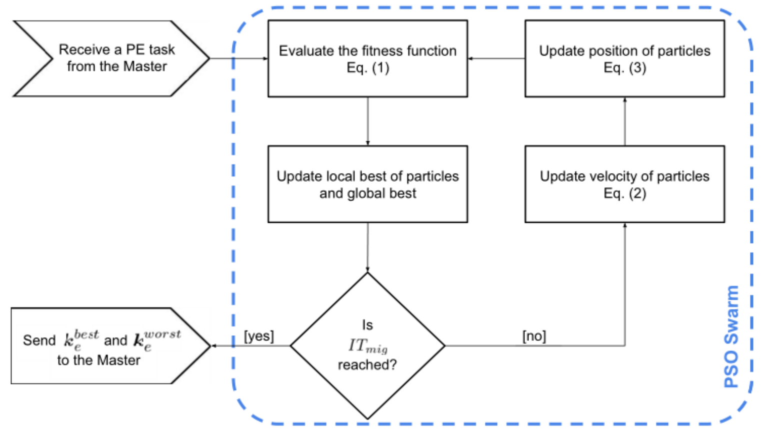
- We exploited such MPI architecture to implement a «divide-et-impera» GA for the **haplotyping problem** (GenHap)
  - The Master **breaks down** the data into Σ **separate subproblems** and sends the data to Σ Slaves
  - Each slave, running on a **separate processor**, performs a **combinatorial optimization** on its data to solve a fragment of the overall problem
  - The master **receives the partial solutions** and eventually **recombines them** into a final optimal haplotype
  - [Tangherloni *et al.*, BMC Bioinform 2019]



Another example: MS^{2}PSO

- Master-slave extension of cuPEPSO
    - **Two-level parallelism**: **MPI** to handle multiple swarms, **GPUs** for biochemical simulation (cupSODA) in each swarm
    - The message sent from the Master is a **PE task**, i.e., the **part of the whole PE** before the next migration
    - The Slave returns the information about the **best particles** in its swarm to distribute the information to the other swarms
    - Each runs on a **separate node** with GPU
    - [Tangherloni *et al.*, Proc PDP 2018]




Receive a PE task from the Master
Evaluate the fitness function Eq. (1)
Update position of particles Eq. (3)
Update local best of particles and global best
Update velocity of particles Eq. (2)
Is IT_{mig} reached?
[yes]
[no]
Send k_e^{best} and k_e^{worst} to the Master
PSO Swarm

Wrapping up

- When multiple objective functions must be optimized simultaneously, multi-objective optimization (MOO) can be leveraged
- MOO can be tackled using both evolutionary (e.g., NSGA and derived algorithms) or swarm intelligence methods (e.g., MOPSO and derived algorithms)
- Population-based heuristics applied to real-world problems are computationally challenging. Fortunately, they can be accelerated using parallel architectures
- Among the options, GPUs provide access to tera-scale performances and thousands cores on common workstations. In order to exploit clusters and supercomputers, MPI can be used
- Talking about real-world problems, the next lecture will focus on applications in Systems Biology, with a special focus to the parameter estimation of biochemical systems