Parallel & Distributed Computing: Lecture 19

Alberto Paoluzzi

December 7, 2017

Parallel computing: Channels

Examples from
[*https://github.com/cvdlab/LARLIB.jl*](https://github.com/cvdlab/LARLIB.jl)

1. Algorithm: spatial arrangement from geometric data
2. Produce & Consume paradigm: Channels in Julia
3. Parallel implementation: Spatial_arrangement in LARLIB.jl

Algorithm: spatial arrangement from geometric data

Arrangement

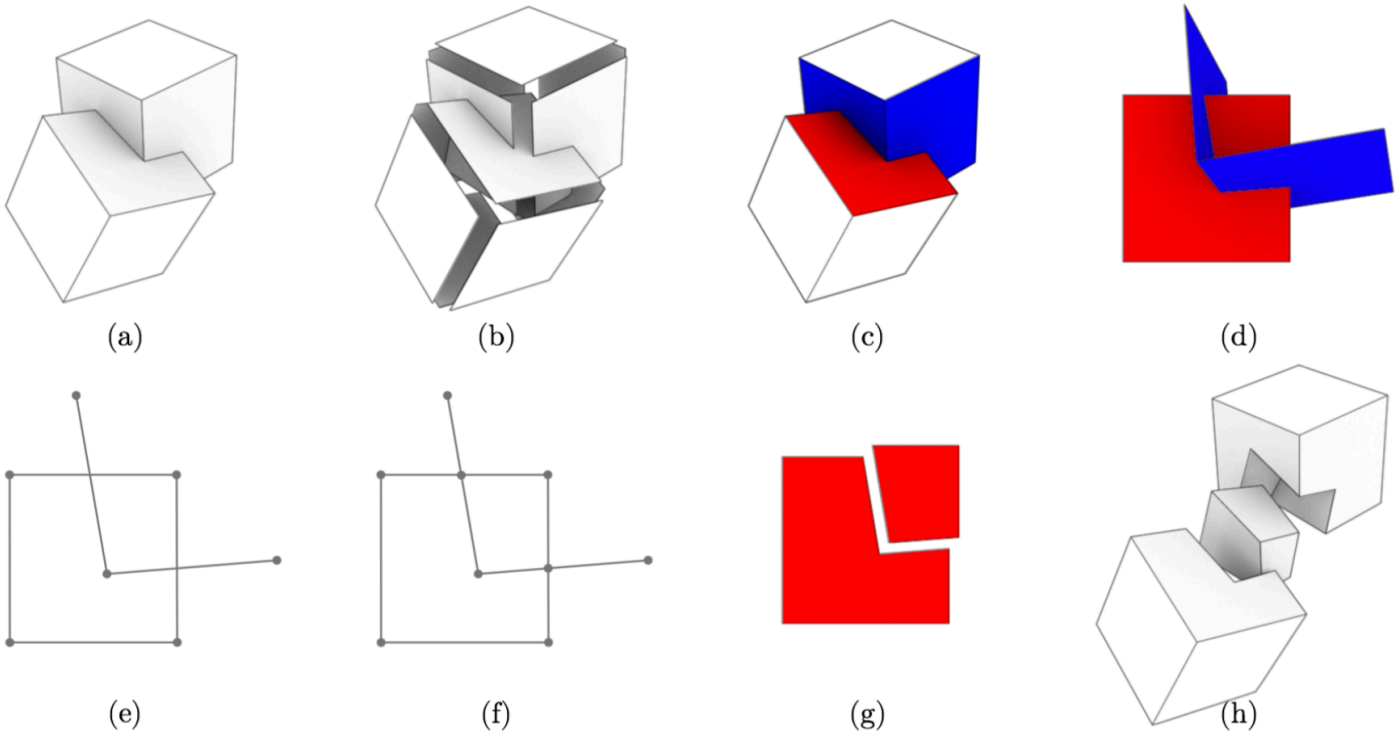

Fig. 1. Cartoon display of the Merge algorithm: (a) the two input solids; (b) the exploded input collection B of 2-cells embedded in $\mathbb{E}^3$; (c) 2-cell σ (red) and the set $\Sigma(\sigma)$ (blue) of possible intersection; (d) affine map of $\sigma \cup \Sigma$ on $z = 0$ plane; (e) reduction to a set of 1D segments in $\mathbb{E}^2$; (f) pairwise intersections; (g) regularized plane arrangement $\mathcal{A}(\sigma \cup \Sigma)$; (h) exploded 3-complex extracted from the 2-complex $X_2(\cup_{\sigma \in B} \mathcal{A}(\sigma \cup \Sigma))$ in $\mathbb{E}^3$. The LAR representation of both the input data (not a complex) and the output data (3-complex with three 3-cells) is given in Appendix A

Figure 1: Arrangement of $\mathbb{E}^3$ generated by a collection of 2-complexes

Arrangement

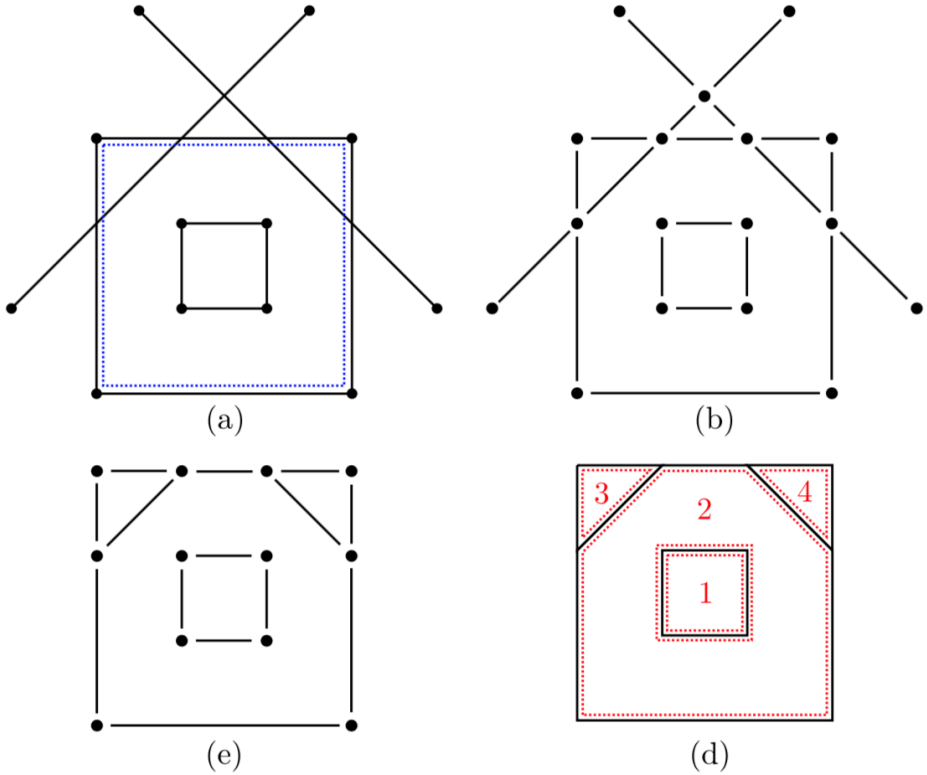

Fig. 2. Basic case: computation of the regularized arrangement of a set of lines in $\mathbb{E}^2$: (a) the input, i.e. the 2-cell σ (blue) and the line segment intersections of $\Sigma(\sigma)$ with $z = 0$; (b) all pairwise intersections; (c) removal of the 1-subcomplex external to σ. To this purpose an efficient and robust point classification algorithm [?] is used, ; (d) interior cells of the regularized 2-complex $X_2 = \mathcal{A}(\sigma \cup \Sigma)$ generated as arrangement of $\mathbb{E}^2$ produced by $\sigma \cup \Sigma$.

Figure 2: Arrangement of a single 2-face in $z = 0$

Arrangement

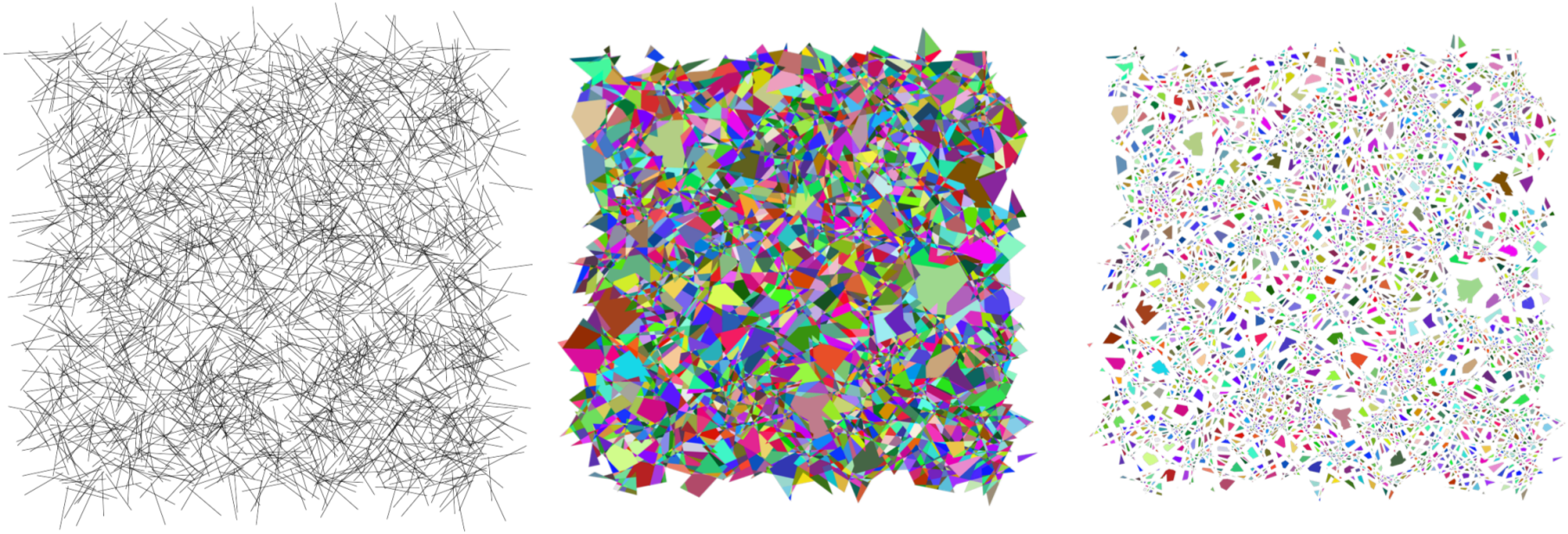

Fig. 5. The regularized 2D arrangement X_2 of the plane generated by a set of random line segments. Note that cells $\sigma \in X_2$ are not necessarily convex. The Euler characteristic is $\chi = \chi_0 - \chi_1 + \chi_2 = 11361 - 20813 + 9454 = 2$.

Figure 3: Arrangement of 2D plane generated by random line segments

Arrangement

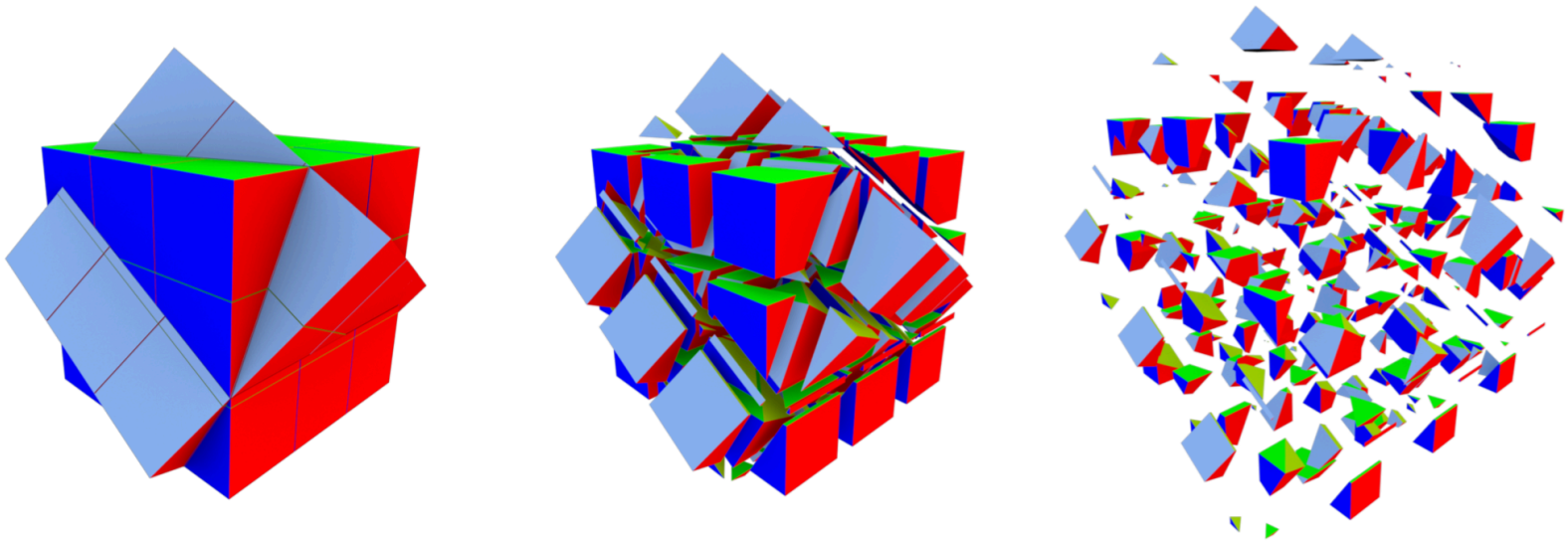

Fig. 6. Merge of two rotated $3 \times 3 \times 3$ cuboidal 3-complexes: (a) the initial positions of the two input 3-complexes, each with 3^3 unit cubic cells; (b) 3-cells of merged complex, with space explosion of small scaling parameter; (c) larger space explosion. Each exploded cell is translated by a scaling-induced movement of its centroid. Output cells are not necessarily convex. In the merged complex we get 236 three-cells and 816 two-cells, both possibly non-convex. Ratio new/old 3-cells is 4.3704.

Figure 4: Arrangement of 3 space generated by two Rubik cubes (2-complexes)

Arrangement

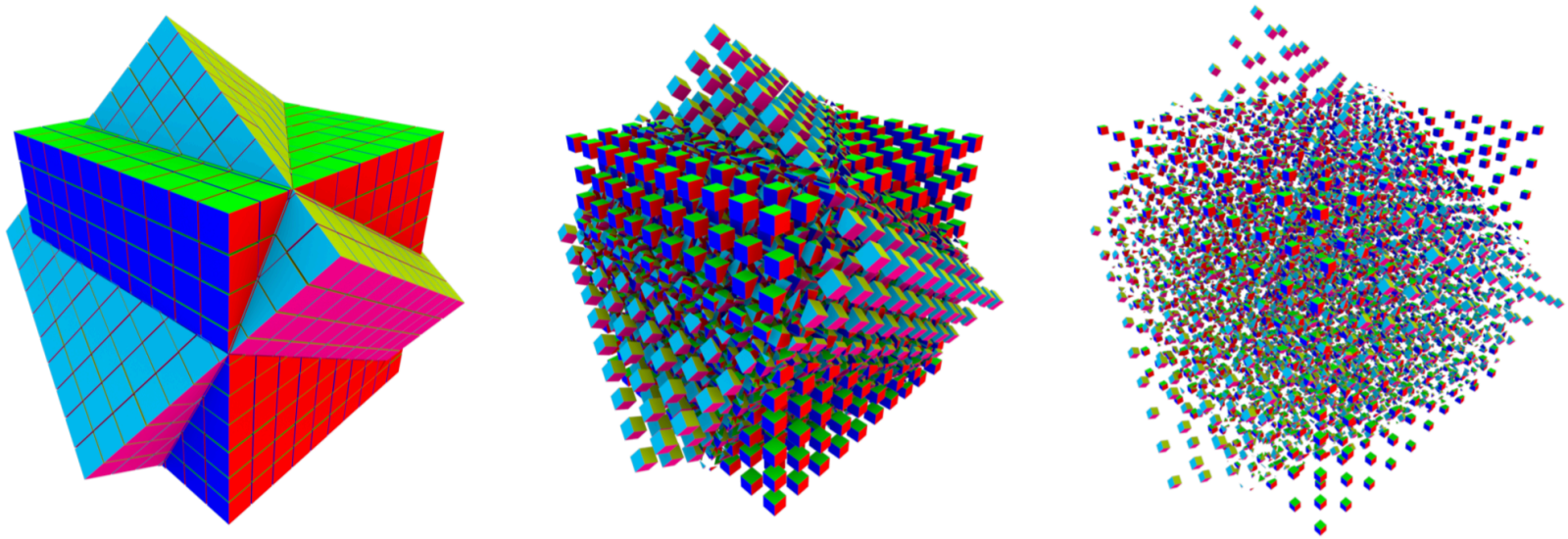

Fig. 7. Merge of two 10^3 3-complexes. In the merged complex we get 8655 3-cells, 26600 2-cells (faces), 26732 1-cells (edges), and 8787 0-cells (vertices). Please note that the Euler characteristic is $\chi = \chi_0 - \chi_1 + \chi_2 - \chi_3 = 8787 - 26732 + 26600 - 8655 = 0$, since the decomposed 3-complex (no its exploded image!) is homeomorphic to the n-sphere, where $\chi = 1 + (-1)^n$, with $n = 3$. Ratio new/old 3-cells is 4.3275.

Figure 5: Arrangement of 3 space generated by two $10 \times 10 \times 10$ cubes (2-complexes)

Algorithm: fragmentation of 2-cells

ALGORITHM 3: *Fragmentation of 2-cells*

Input: $\mathcal{S}_2 \subset \mathcal{S}_{d-1}$ # *collection* of all 2-cells from $\mathcal{S}_{d-1}$ input in $\mathbb{E}^d$
Output: $[\partial_2]$ # CSC signed matrix
$\widetilde{\mathcal{S}_2} = \emptyset$ # initialisation of collection of local fragments
for $\sigma \in \mathcal{S}_2$ **do** # for each 2-cell σ in the input set
  $M = SubManifoldMap(\sigma)$ # affine transform s.t. $\sigma \mapsto x_3 = 0$ subspace
  $\Sigma = M\,\mathcal{I}(\sigma)$ # apply the transformation to (possible) incidencies to σ
  $\mathcal{S}_1(\sigma) = \emptyset$ # collection of line segments in $x_3 = 0$
  for $\tau \in \Sigma$ **do** # for each 2-cell τ in Σ
    $\mathcal{P}(\tau), \mathcal{L}(\tau) = \emptyset, \emptyset$ # intersection points and int. segment(s) with $x_3 = 0$
    for $\lambda \in X_1(\tau)$ **do** # for each 1-cell λ in $X_1(\tau)$
      if $\lambda \not\subset \{\mathbf{q} \,|\, x_3(\mathbf{q}) = 0\}$ **then** $\mathcal{P}(\tau)$ += $\{\mathbf{p}\}$ # append the intersection point of λ with $x_3 = 0$
    end
    $\mathcal{L}(\tau) = Points2Segments(\mathcal{P}(\tau))$ # Compute a set of collinear intersection segments
    $\mathcal{S}_1(\sigma)$ += $\mathcal{L}(\tau)$ # accumulate intersection segments with σ generated by τ
  end
  $X_2(\sigma) = \mathcal{A}(\mathcal{S}_1(\sigma))$ # arrangement of σ space induced by a soup of 1-complexes
  $\widetilde{\mathcal{S}_2}$ += $M^{-1}\,X_2$ # accumulate local fragments, back transformed in $\mathbb{E}^d$
end
$[\partial_1] = QuotientBases(\widetilde{\mathcal{S}_2})$ # identification of 0- and 1-cells using *kd*-trees and canonical LAR
$[\partial_2] = Algorithm\_1([\partial_1])$ # output computation GRANDE STRONZATA
return $[\partial_2]$

Figure 6: Reconstruction of ∂_2 matrix

Algorithm: Reconstruction of ∂_3 matrix

ALGORITHM 1: *Computation of signed* $[\partial_d^+]$ *matrix*

```
/* Pre-condition: d equal to space dimension, s.t. (d − 1)-cells are shared by two d-cells */
/*                                                                                        */
Input: [∂_{d−1}]  # CSC signed matrix
Output: [∂_d^+]  # CSC signed matrix
m, n = [∂_{d−1}].size; marks = Zeros(n)  # initializations
while Sum(marks) < 2n do
    σ = Choose(marks)        # select the (d − 1)-cell seed of the column extraction
    if marks[σ] == 0 then [c_{d−1}] = [σ]
    else if marks[σ] == 1 then [c_{d−1}] = [−σ]
    [c_{d−2}] = [∂_{d−1}] [c_{d−1}]        # compute boundary c_{d−2} of seed cell
    while [c_{d−2}] ≠ [] do        # loop until boundary becomes empty
        corolla = []
        for τ ∈ c_{d−2} do        # for each hinge τ cell
            [b_{d−1}] = [τ]^t [∂_{d−1}]        # compute the τ coboundary
            pivot = {|b_{d−1}|} ∩ {|c_{d−1}|}        # compute the τ support
            if τ > 0 then adj = Next(pivot, Ord(b_{d−1}))        # compute the new adj cell
            else if τ < 0 then adj = Prev(pivot, Ord(b_{d−1}))
            if ∂_{d−1}[τ, adj] ≠ ∂_{d−1}[τ, pivot] then corolla[adj] = c_{d−1}[pivot]        # orient adj
            else corolla[adj] = −(c_{d−1}[pivot])
        end
        [c_{d−1}] += corolla        # insert corolla cells in current c_{d−1}
        [c_{d−2}] = [∂_{d−1}] [c_{d−1}]        # compute again the boundary of c_{d−1}
    end
    for σ ∈ c_{d−1} do marks[σ] += 1        # update the counters of used cells
    [∂_d^+] += [c_{d−1}]        # append a new column to [∂_d^+]
end
return [∂_d^+]
```

Figure 7: Reconstruction of ∂_3 matrix

Produce & Consume paradigm: Channels in Julia

Definition: produce and consume problem

- Also known as the bounded-buffer problem is a classic example of a multi-process synchronization problem.

Definition: produce and consume problem

- Also known as the bounded-buffer problem is a classic example of a multi-process synchronization problem.
- Two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue (FIFO)

Definition: produce and consume problem

- Also known as the bounded-buffer problem is a classic example of a multi-process synchronization problem.
- Two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue (FIFO)
- The producer's job is to generate data, put it into the buffer, and start again.

Definition: produce and consume problem

- Also known as the bounded-buffer problem is a classic example of a multi-process synchronization problem.
- Two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue (FIFO)
- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Solution: produce and consume problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full.

Solution: produce and consume problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.

Solution: produce and consume problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Solution: produce and consume problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- The solution can be reached by means of inter-process communication, typically using semaphores.

Solution: produce and consume problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- The solution can be reached by means of inter-process communication, typically using semaphores.
- An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Solution: produce and consume problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- The solution can be reached by means of inter-process communication, typically using semaphores.
- An inadequate solution could result in a deadlock where both processes are waiting to be awakened.
- The problem can also be generalized to have multiple producers and consumers.

Channel in Julia

- A channel can be visualized as a pipe, i.e., it has a write end and read end.

Channel in Julia

- A channel can be visualized as a pipe, i.e., it has a write end and read end.
- Multiple writers in different tasks can write to the same channel concurrently via `put!` calls.

Channel in Julia

- A channel can be visualized as a pipe, i.e., it has a write end and read end.
- Multiple writers in different tasks can write to the same channel concurrently via `put!` calls.
- Multiple readers in different tasks can read data concurrently via `take!` calls.

Channel operations 1/2

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of `type T`.

Channel operations 1/2

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of `type T`.
  - If the type is not specified, the channel can hold objects of any type.

Channel operations 1/2

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of `type T`.
  - If the type is not specified, the channel can hold objects of any type.
  - szrefers to the maximum number of elements that can be held in the channel at any time.

Channel operations 1/2

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of `type T`.
  - If the type is not specified, the channel can hold objects of any type.
  - szrefers to the maximum number of elements that can be held in the channel at any time.
  - For example, `Channel(32)` creates a channel that can hold a maximum of 32 objects of any type.

Channel operations 1/2

- Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of `type T`.
  - If the type is not specified, the channel can hold objects of any type.
  - szrefers to the maximum number of elements that can be held in the channel at any time.
  - For example, `Channel(32)` creates a channel that can hold a maximum of 32 objects of any type.
  - A `Channel{MyType}(64)` can hold up to 64 objects of MyType at any time.

Channel operations 2/2

- If a Channel is empty, readers (on a `take!` call) will block until data is available.

Channel operations 2/2

- If a Channel is empty, readers (on a `take!` call) will block until data is available.
- If a Channel is full, writers (on a `put!` call) will block until space becomes available.

Channel operations 2/2

- If a Channel is empty, readers (on a `take!` call) will block until data is available.
- If a Channel is full, writers (on a `put!` call) will block until space becomes available.
- `isready` tests for the presence of any object in the channel, while `wait` waits for an object to become available.

Channel operations 2/2

- If a Channel is empty, readers (on a `take!` call) will block until data is available.
- If a Channel is full, writers (on a `put!` call) will block until space becomes available.
- `isready` tests for the presence of any object in the channel, while `wait` waits for an object to become available.
- A Channel is in an open state initially. This means that it can be read from and written to freely via `take!` and `put!` calls.

Channel operations 2/2

- If a Channel is empty, readers (on a `take!` call) will block until data is available.
- If a Channel is full, writers (on a `put!` call) will block until space becomes available.
- `isready` tests for the presence of any object in the channel, while `wait` waits for an object to become available.
- A Channel is in an open state initially. This means that it can be read from and written to freely via `take!` and `put!` calls.
- `close` closes a Channel. On a closed Channel, `put!` will fail.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.
- Finally all the results are printed out.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.
- Finally all the results are printed out.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.
- Finally all the results are printed out.

```
julia> const jobs = Channel{Int}(32);

julia> const results = Channel{Tuple}(32);

julia> function do_work()
           for job_id in jobs
               exec_time = rand()
               sleep(exec_time) # simulates elapsed time doing actual work
                                # typically performed externally.
               put!(results, (job_id, exec_time))
           end
       end;
```

Example from Julia docs

- Consider a simple example using channels for inter-task communication.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.
- Finally all the results are printed out.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.
- Finally all the results are printed out.

Example from Julia docs

- Consider a simple example using channels for inter-task communication.
- We start 4 tasks to process data from a single jobs channel.
- Jobs, identified by an id (job_id), are written to the channel.
- Each task in this simulation reads a job_id, waits for a random amout of time and writes back a tuple of job_id and the simulated time to the results channel.
- Finally all the results are printed out.

```
julia> n = 12;

julia> @schedule make_jobs(n); # feed the jobs channel with "n" jobs

julia> for i in 1:4 # start 4 tasks to process requests in parallel
           @schedule do_work()
       end

julia> @elapsed while n > 0 # print out results
           job_id, exec_time = take!(results)
           println("$job_id finished in $(round(exec_time,2)) seconds")
           n = n - 1
       end
```

Parallel implementation: Spatial_arrangement in LARLIB.jl

define channels

```
function frag_face_channel(in_chan, out_chan, V, EV, FE, sp_idx)
    run_loop = true
    while run_loop
        sigma = take!(in_chan)
        if sigma != -1
            put!(out_chan, frag_face(V, EV, FE, sp_idx, sigma))
        else
            run_loop = false
        end
    end
end
function frag_face(V, EV, FE, sp_idx, sigma)
    vs_num = size(V, 1)

    sigmavs = (abs.(FE[sigma:sigma,:])*abs.(EV))[1,:].nzind
    sV = V[sigmavs, :]
    sEV = EV[FE[sigma, :].nzind, sigmavs]

    M = submanifold_mapping(sV)
    tV = ([V ones(vs_num)]*M)[:, 1:3]
```

spatial_arrangement

```
function spatial_arrangement(V::Verts, EV::Cells, FE::Cells; multiproc=fals
    fs_num = size(FE, 1)
    sp_idx = spatial_index(V, EV, FE)

    rV = Verts(0,3)
    rEV = spzeros(Int8,0,0)
    rFE = spzeros(Int8,0,0)

    if (multiproc == true)
        in_chan = RemoteChannel(()->Channel{Int64}(0))
        out_chan = RemoteChannel(()->Channel{Tuple}(0))

        @schedule begin
            for sigma in 1:fs_num
                put!(in_chan, sigma)
            end
            for p in workers()
                put!(in_chan, -1)
            end
        end
```

spatial_arrangement

```
    else
        for sigma in 1:fs_num
            print(sigma, "/", fs_num, "\r")
            nV, nEV, nFE = frag_face(V, EV, FE, sp_idx, sigma)
            rV, rEV, rFE = skel_merge(rV, rEV, rFE, nV, nEV, r
        end

    end

    rV, rEV, rFE = merge_vertices(rV, rEV, rFE)
```

Look at the code

https://github.com/cvdlab/LARLIB.jl/blob/master/src/spatial_arrangement.jl