Mission Briefing: The Pincer Pox Pandemic

Note: You must register for the project by **Tuesday, 27.06.2023, 23:59**, see page 7 for details.
If you want to use milestone A, it needs to be handed in by **Monday, 03.07.2023, 23:59**.
The submission deadline for milestone Ω is **Monday, 24.07.2023, 23:59**.

1 Context

Less than four years after the global pandemic, another threat emerges: *pincer pox*. You can find details about this less dangerous, but more rave-y virus in appendix A.

In understanding a viral infection, the two most important scientific disciplines are *virology* and *epidemiology*. Virology is the study of viruses themselves and their interaction with their hosts. Epidemiology takes a more zoomed out perspective and studies the spread of diseases within populations.

One of the methods used in epidemiology is *modelling*: Scientists create abstract models of society and a disease to simulate its spread, allowing them to analyse the effect of certain changes to the outcome. There are different ways of creating such a model. Some are more abstract, describing the state of the system with a few variables, interconnected by differential equations. Other approaches are more detailed, trading higher accuracy for higher computational costs.

One of the more detailed approaches is agent-based modelling, an approach where the population is described as a set of individual *agents*, an abstract model of humans (or other animals in question). Then, a computer can simulate the behaviour of the whole model simply by simulating all agents. Depending on the size of the model, this can require simulating millions of agents over an extended period of time.

In this project, your mission is to help predict (and thus, indirectly, curb) the spread of pincer pox using agent-based modelling.

2 An Agent-Based Model

We consider a simplistic model. It works as follows:

Time is measured in an abstract unit called *ticks*.

Space is two-dimensional[1], split into cells by an equidistant grid. The coordinates are zero-based, starting in the top-left corner. Figure 1 shows an example of a 20 × 10 space.

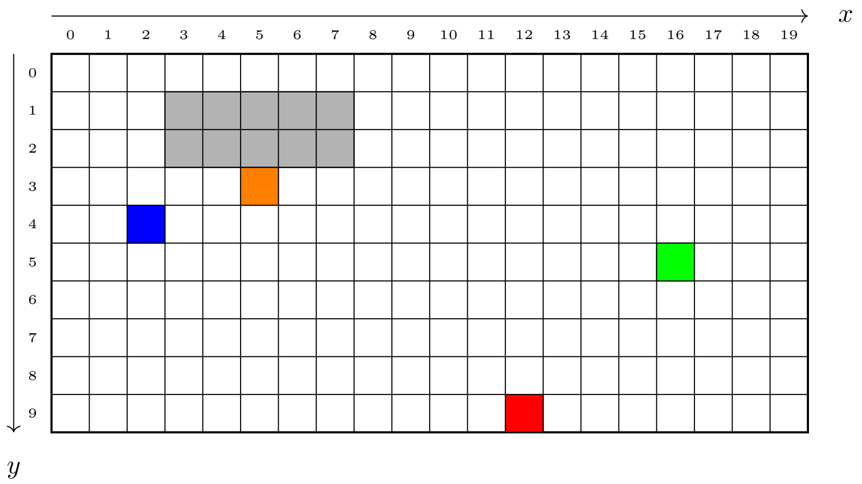

Figure 1: A sample space with one obstacle.

[1]Conspiracy theorists rejoice!

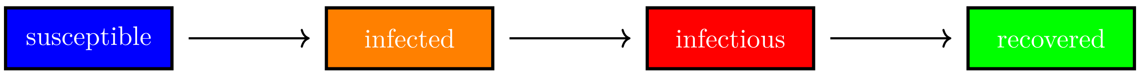

Figure 2: The states of the infection.

Obstacles can occupy rectangular regions in space. The example in Figure 1 has one obstacle. Its top left cell has coordinates $(3, 1)$ and it is 5 cells wide and 2 cells tall.

A Person occupies one cell of space. A person can only be in a space that is not occupied by an obstacle or another person. To randomise its behaviour, every person has a pseudorandom number generator. Each tick, a person can move up to one cell vertically, horizontally, or diagonally, or stay in place. The movement is calculated each tick using inertia (a person tends to keep moving in the direction it is moving in) and its pseudorandom number generator. All persons move in the order they are specified in. If a move would place a person in an invalid position (a cell that is obstructed, occupied, or outside the space), the move is not executed and the person's velocity is reset. A position is occupied if a person has been moved on it in the current tick or a person has been on the position at the beginning of the current tick, prior to being moved. We call the position the person has been at the beginning of the current tick, the person's *ghost position.*

The Infection is modelled by every person having one of four states, as shown in Figure 2. How a person transitions to the next state depends on the state:

- A person becomes *infected* if it was *susceptible* in the previous tick, it is currently breathing in according to its pseudorandom number generator, and, after all moves have been processed, there is another person that
  - (a) is infectious,
  - (b) is at most `infectionRadius` cells away (where the total distance is measured as the sum of horizontal and vertical distances), and
  - (c) is currently coughing according to its pseudorandom number generator.

  These conditions are evaluated *after* all other state transitions have been executed.
- A person becomes *infectious* if it has been *infected* for `incubationTime` ticks.
- A person becomes *recovered* if it has been *infectious* for `recoveryTime` ticks.

Parameters

The behaviour of the system can be customised by these parameters:

Name	**Description**
`coughThreshold`	modifies how often a person coughs
`breathThreshold`	modifies how often a person breathes in
`accelerationDivisor`	modifies how likely a person is to accelerate
`recoveryTime`	the number of ticks a person is infectious before recovering
`infectionRadius`	the maximum (non-diagonal) distance the infection can spread directly
`incubationTime`	the number of ticks a person is infected before becoming infectious

Outputs

The simulation outputs two kinds of data:

- The model of a simulation scenario can contain a set of *statistics queries* in terms of rectangular regions of space. For each of these queries, the output contains an array that describes the number of susceptible, infected, infectious, and recovered persons within the area for every tick of the simulation.
- Optionally (at the cost of slowing down simulation), the simulation can also output a complete description of the population for every tick.

Determinism

While the model describes agents using randomness, we want the simulation to produce deterministic results. The first motivation for this is to allow easy testing. Moreover, it lays the foundation for parallelisation, as it ensures that multiple threads can potentially simulate the behaviour of the same person and get the same result.

Determinism is mainly reached by ensuring the pseudorandom generator of each person behaves deterministically. This is done by specifying the initial state of the generator within the model for each agent, deriving the required

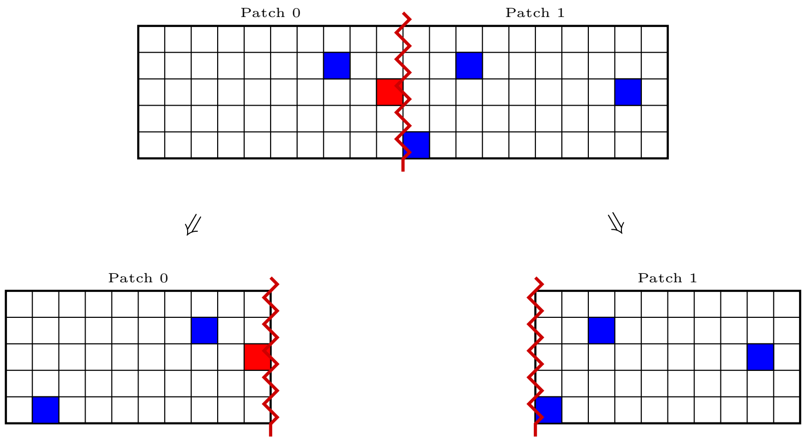

Figure 3: Parallelising looks easy at first glance.

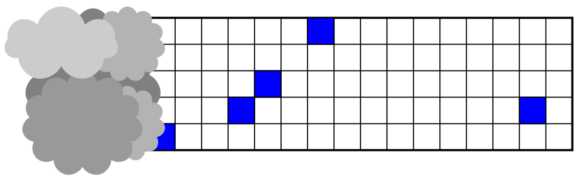

Figure 4: A model with unknown parts.

properties (e. g., whether the agent is coughing) deterministically from this state, and computing the next state in a deterministic but pseudo-random way.

For determinism to work, a few other details have to be taken into account. Most notably, the movement of all persons that may influence each other must be computed in the order they are defined.

Parallelisation

Parallelising such a simulation is nontrivial. At first, the solution may appear easy: We simply cut the space into multiple areas (called *patches*) that can then be handled by one thread each. For simplicity, in this document, we only consider cutting in one dimension. In reality, we use cuts in two dimensions (forming a table-like structure), and enumerate patches left-to-right and top-to-bottom, i. e., the top-left patch has the id 0, it's right neighbour has the id 1, and so on. The idea is demonstrated in Figure 3.

Simply splitting the model like this and them simulating each patch individually is, of course, not correct, as it prevents the parts of the model from interacting with each other. In the example shown in Figure 3, simply splitting the model prevents the infection from ever reaching the right half of the simulated space.

If proper care is taken to consider all ways in which the patches can influence each other (e. g., persons crossing the boundary or infections spreading across the boundary), this problem can be solved by having each patch consider the relevant persons in other (adjacent) patches.

However, doing so creates another problem: This requires patches to communicate at every tick, as computing the behaviour of persons in a patch requires data from other patches.[2] This creates enormous communication overhead, *especially* if the number of patches is high.

To resolve this problem, we need to consider how uncertainty spreads throughout our model. Figure 4 shows a model where a small part on the left is not visible. Let's assume that `infectionRadius` and `incubationTime` are both set to 3. It's clear that the person on the right cannot be affected by anything in the covered area of space in the next tick. However, over time, the area potentially affected by the uncertainty grows: Maybe a person moves out of the unknown area to the right, or a hidden infectious person infects the susceptible person on the left of the visible area, causing the infection to spread towards the right.

This spread of uncertainty has limited speed. For movement, uncertainty grows with one cell per tick, as that is the maximal movement speed of a person. For the infection, the answer is slightly more involved and depends

[2]The same problem surfaces if instead of splitting space into patches assigned to different threads, the population is divided into groups of people assigned to threads.

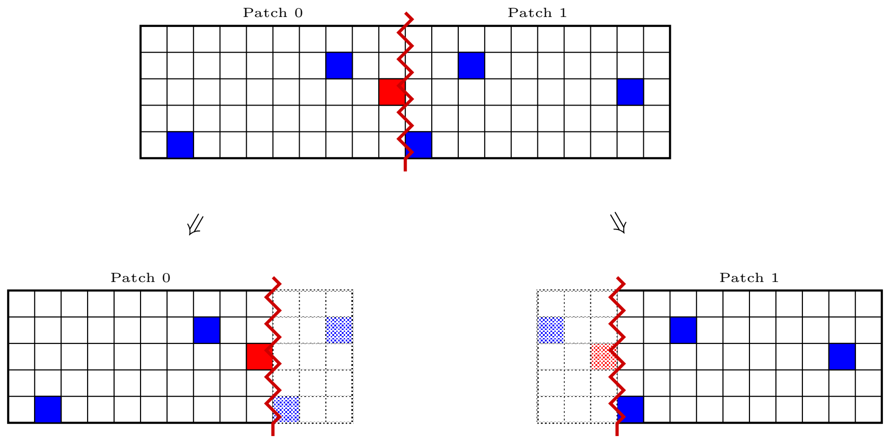

Figure 5: Parallelising correctly and efficiently requires considering the neighbouring area.

on the model parameters. In a single step, the infection can jump `infectionRadius` cells far, as this is the maximum distance the infection can jump over. However, before the infection can make another jump, the newly infected person must become infectious, which takes `incubationTime` ticks. This severely limits the speed at which the infection can spread. Combining movement and infection spread: In a single tick, a person can infect a person that is `infectionRadius` + 1 cells away by moving one step towards it, and then infecting it.

Once we have determined how far uncertainty spreads in our model, we can parallelise computations much more efficiently by including *padding*. The principle behind this is sketched in Figure 5.

While the space is still partitioned into patches (that do not overlap and cover the whole space), every thread not only stores the data for its patch, but also for an area around it, the *padding*. Then, every thread simulates the behaviour of its patch, and the padding, for multiple ticks without communicating with other threads in any way. Due to determinism, both threads will yield the same result, except for the error caused at the outer edge of padding. This error spreads inwards at a fixed pace. After a precomputed number of ticks k that ensure the error cannot cross the boundary between padding and patch, the threads have to synchronise: Every thread discards the data from its padding (as it may be incorrect), then every thread receives the data for its padding area from the respective owners. The phase between this synchronisation is called a *cycle*.

The number of ticks k depends on the padding size and model parameters (e. g., the infection radius). It is a constant that can be computed once at the beginning of simulation.

By ensuring that the threads can compute multiple ticks without any communication, the synchronisation overhead is reduced. This generally increases performance. (A side effect is that some threads can proceed faster than others, advancing the simulation for multiple ticks beyond their neighbours. This is not a problem—the threads will have to wait for each other to advance beyond the kth tick.)

Moreover, synchronisation is *local*: Instead of threads having to wait for a global lock or something similar, they only need to communicate with their (possibly non-direct) neighbours. While this may not be overly important when running on a single, regular computer, this design also allows parallelising over multiple machines, where communication becomes more costly.

The need for synchronisation can be reduced even further by taking obstacles into account: If parts of the padding area or the border area of a patch are blocked by obstacles, ensuring that neighbouring patches cannot influence each other, the threads owning these patches do not need to synchronise. To pass the project, it suffices to use the method `mayPropagateFrom` of the class `com.pseuco.cp23.simulation.common.Utils` to check whether information might propagate from the padding inside the patch area. In case of Rust, the function `may_propagate_from` of the module `simulation` of the crate `spread-sim-core` provides equivalent functionality.

Once every patch has reached the final tick of the simulation, the relevant output needs to be computed. This can be done in two ways:

(a) All threads store all relevant data until the simulation is finished. Then, all data is collected and merged by a central management thread.

(b) All threads submit the relevant data to a central manager during simulation. This allows for more parallelisation, but this data submission must not become a bottleneck: The threads doing the simulation may not wait for a condition to submit their data, and care has to be taken that submission only rarely has to wait for a lock.

Assignments

Assignment 1: Concurrent Pandemic Simulations

Your assignment is to parallelise a sequential implementation of the agent-based simulation using the approach described above. The partitioning of the space into patches is fixed in the scenario. The same holds for the padding size. The number of ticks you can compute concurrently without the simulation becoming incorrect is not given and needs to be computed by you.[3]

Ensure that threads synchronise only as much as is necessary, exploiting the presence of obstacles if possible. During your simulation, your threads may only synchronise with their (possibly non-direct) neighbours (as defined by the partitioning and padding size), i. e., *not* wait for global data structures or a central thread managing the simulation. As described above, it is allowed to submit results to a central management thread to compute the output concurrently. However, you need to ensure that submitting threads rarely have to wait for a lock and not use any conditions in case you use this approach.

This assignment is mandatory to pass the project.

Assignment 2: Advanced Concurrency

After having parallelised the simulation as described above, take a look at avenues for improvement. Implement an alternative approach that produces (correct) results faster. For this exercise, you may deviate from the partitioning specified in the scenario or use a completely different approach for parallelisation. You should, however, still try to minimise unnecessary communication and reliance on global data structures during simulation.

This assignment is optional. However, it is mandatory to qualify for the bonus.

[3]If this number is 0, you must throw an `InsufficientPaddingException` in Java and return an `InsufficientPaddingError` in Rust.

Formalities

Groups & Registration

There are three steps you must take **by Tuesday, 27.06.2023, 23:59** to register for the project:

(a) The project is solved in groups of 2 students.[4] Please find another student to work on the project with. If necessary, you can use the partner market on our discussion board to find a project partner.

Then, form a project team in dCMS:
`https://dcms.cs.uni-saarland.de/cp_23/landing#teams`

Should you be unable to follow these steps (for example because you cannot find a partner), but want to participate in the project, you must contact us via a message to **@Organizers** on our discussion board by Tuesday, 27.06.2023, 23:59.

(b) Submission of the project will be handled via Git. Register at our GitLab Instance, if you don't have an account yet, and enter your username **by Tuesday, 27.06.2023, 23:59** on your personal status page in dCMS. To make sure there are no typos, after saving, please click your dGit username on your personal status page. This should open your profile in dGit.

(c) Please choose whether you want to complete the project in Java or in Rust and submit this choice on your personal status page in dCMS.

This choice is preliminary. We use it to determine which template we initialize your project repository with and which tutor we assign you to. If you wish to switch programming languages afterwards, please contact us via a message to **@Organizers** on our discussion board to discuss the best path forward.

It's sufficient if one project partner does this. (If both project partners answer, and their answers differ, we will make a nondeterministic choice.)

> Carefully read these instructions to ensure you do not miss a step. If you do not form and register a group or fail to supply a valid GitLab username on your dCMS profile, no project repository will be created for you and you will be unable to participate in the project.

Solutions and Milestones

Your solution must include:

- Your implementation in the form of a Java 20 or Rust project.[5] Ensure your code is properly commented and overall self-explanatory.
- Documentation ($\leq$ 9.000 non-whitespace characters) that explains your solution.

  In this document, focus on describing the details of how you parallelise the work, and why your use of concurrency is correct. For every part of your code that uses concurrency, make clear why there are no data races, and whenever you use locks, explain how you ensure no deadlock can occur.

  The document should also state and briefly justify how you calculate the number of ticks that can be computed without synchronisation.

  Please use the Wiki of your GitLab repository for this documentation. *We cannot consider other documents placed in your repository.* Please call the Wiki page we should read *home*.

We strongly recommend that you submit a first draft of this document as your *milestone A* that explains how you intend to solve the project by Monday, 03.07.2023, 23:59. For this milestone, it is not required that you argue why your solution is correct.

Submitting milestone *A* is not mandatory. You can skip it *at your own risk*. If you decide to do so, please read on to ensure you understand the consequences.

[4]There will be at most one exception to this rule.

[5]For our tests, we use the pre-build OpenJDK Adoptium® Temurin™ and Rust `nightly-2023-06-23`. The nightly Rust version is required because our test infrastructure uses some unstable features. Note that you should not use any unstable features yourself as they may contain soundness holes, i.e., may render Rust's safety guarantees void. As long as you do not explicitly enable any unstable features yourself, you may argue for the correctness of your code based on Rust stable 1.70. The file `rust-toolchain.toml`, shipped with the project template, takes care of configuring your development environment (e.g., `cargo` should automatically use the correct version).

The end of the project forms *milestone* Ω, to be handed in by Monday, 24.07.2023, 23:59. To grade this milestone, we will consider the code in your repository and the Wiki page called *home* at the point in time when the deadline passes.

Milestone *A* serves two purposes:

- It provides you important feedback on your intended solution before you have to implement it. Your tutor will check your submission and look for flaws in your plan. If necessary, your tutor will offer you a mentoring appointment to discuss any problems and potential solutions.

  Make sure your milestone *A* describes your plans as accurately and detailed as possible to ensure you get relevant feedback.

- We provide a safety net to groups who use milestone *A*. Provided you

  (a) submit a milestone *A* that documents a serious attempt to understand and solve the problems posed by the assignment,

  (b) participate in the mentoring appointment if offered by your tutor, and

  (c) submit a milestone Ω that documents a serious attempt to implement a working solution to the problem posed by the assignment,

  you will be given a second attempt to submit a correct milestone Ω should your first submission not pass.

  If you do not satisfy the requirements above and your first submission for milestone Ω does not pass, you have failed the project.

Automatic Testing

We will provide tests for the project, both in the form of tests made available for you to run yourself and in the form of tests run automatically server-side. We provide these on a best-effort basis and cannot guarantee their correctness or availability.

Also, keep in mind that while test failures indicate that there is a problem with your solution (or a bug in our test), **passing our tests does not indicate the absence of problems**. This is not only because in the presence of nondeterministic bugs, tests can always be passed by chance or by implementation detail, but because some requirements we pose are not testable in general, for example the absence of data races, busy waits, or other conceptual problems of concurrency. These will be graded manually.

Make sure you read the requirements for passing further below instead of blindly relying on tests.

Defence

After handing in milestone Ω and it being graded, you will be required to defend your project in a short in-person meeting with us. These meetings are currently scheduled for 02.08.2023. If, for good reasons (illness, other university-related events, et cetera), you will not be available, you can make a different appointment with us.

Choice of Programming Language

We give your the option of completing this project either in Java or in Rust.

Please note that completing the project in Rust required more knowledge of Rust than has been covered in the lecture. We recommend choosing Rust only if you either are already familiar with Rust, or are interested and willing to spend additional time to sharpen your Rust skill on your own.

Reference Implementation

We are providing a sequential reference implementation in both Java and Rust that you can (and must) use as the basis for your implementation. It also serves as the exact specification of the input and output formats and the simulation algorithm. The Git repository we provide you with will already be initialised with this implementation (in the programming language you have chosen while registering for the project).

For Java, your implementation should go into the package `com.pseuco.cp23.simulation.rocket`. For Rust, it should go into the crate `spread-sim-rocket`. **You are not allowed to modify any other files outside of this package as they will be replaced by our testing infrastructure.** It is not necessary to modify any of the existing files, if you nevertheless feel the need to modify them, copy them to the rocket package or crate.

For Java, you should implement assignment 1 in the class `Rocket` and assignment 2 in the class `Starship`. For Rust, the `launch` function must be implemented for both assignment 1 and 2. Please do not change the signature of these classes and functions as we rely on their signature and them implementing the `Simulation` interface (Java only) for our tests.

The reference implementation parses the command line arguments for you. It accepts the following command line arguments:

Name	Description
`--slug`	use the slow sequential implementation for simulation
`--rocket`	use your concurrent implementation (for assignment 1) for simulation
`--starship`	use your concurrent implementation (for assignment 2) for simulation
`--padding`	the padding that should be used
`--scenario`	the scenario file containing the initial state of the model
`--out`	the file the output should be written to

Our framework contains Javadoc/Rustdoc comments that can be built with Gradle/Rustdoc. You can find details on that in the `README` file provided with the reference implementation.

We are also providing a testing and visualisation service available online that you can use to validate and visualise your output. You can find it at `https://missioncontrol.pseuCo.com/`.

Validation Interface

To make the project easier to debug and test, our reference implementation includes a validation interface. It contains methods your code needs to call at specific points during its execution. The performance of your solution will be evaluated with a no-op validator that does nothing. For automated testing, we will also use the validation interface to influence and validate the behaviour of your program for compliance with the requirements described in the program statement.

Your solution is required to call the validator as specified in its documentation. Make sure to read the documentation of the validation interface for details.

Passing Criteria

To pass the project, you need to fulfil the following criteria:

- Your program must solve assignment 1 correctly.
- Your program needs to work concurrently (with adequate efficiency).
- You must convince us that you have created your solution together with your teammate.
- You must convince us that your program is guaranteed to work correctly.

For Java: To satisfy the last requirement, you can either use what you have learned in our lecture, or argue with the Java Language Specification in version 20.[6]

For Rust: To satisfy the last requirement, you can argue with the documentation of Rust's standard library and the documentation of the third-party libraries shipped with the template. If you use any other third-party libraries, you take responsibility for any (concurrency) bugs within them.

Bonus

You can receive a bonus (of 0.3 / 0.4 on your final grade of the module) if you solve both assignments 1 and 2.

We will select the groups that receive this bonus based on the following criteria:

- effective parallelisation,
- efficient and elegant implementation, and
- exemplary documentation.

Typically, about 3 to 5 groups receive this bonus.

[6]It is not sufficient to argue that the structures you use are *thread safe* according to their documentation as that is a promise without a precisely defined formal meaning. Also keep in mind that you cannot argue using the concrete implementation of the Java standard classes you use as your implementation is required to work correctly under any (correct) Java runtime.

A Pincer Pox

Pincer pox is am infectious disease that affects both crustaceans and humans, with transmission occurring when a human is pinched by an infected crustaceans. Furthermore, it is transmissible between humans through airborne particles, making it a significant public health concern. The disease is characterized by a distinctive symptom wherein infected individuals exhibit behaviors reminiscent of a crab's distinctive movement, commonly referred to as "raving like a crab."

A.1 Causes and Transmission

Pincer pox is caused by a viral pathogen belonging to the Pincerovirus family, which affects the nervous system of its hosts. When a human is pinched by an infected crustacean (most often a crab), the virus is introduced into the bloodstream, where it begins to replicate and spread throughout the body. Additionally, the airborne transmission occurs when infected individuals, particularly those exhibiting symptoms, release virus-laden respiratory droplets into the surrounding environment, which can be inhaled by others.

A.2 Symptoms

The primary symptom of pincer pox is the transformation of infected individuals into temporary "crab-like" ravers. Affected humans or crustacean typically exhibit an altered gait, characterized by sideways movement and involuntary, exaggerated movements of the extremities that even in humans resemble a crab's pincer motion. This peculiar behavior, often accompanied by spontaneous clapping, has led to the disease's colloquial association with the popular "crab rave" song. The duration of this crab-like behavior varies among individuals, typically lasting for a few hours to a couple of days before subsiding.

In addition to the crab-like movements, infected individuals may experience mild flu-like symptoms, such as fever, fatigue, and body aches. However, these symptoms are generally short-lived and resolve on their own without requiring specific medical intervention.

Figure 6: An infected *Dungeness crab*, exhibiting the typical sideways movement

A.3 Diagnosis and Treatment

Diagnosing pincer pox primarily relies on clinical observation of the distinct crab-like raving behavior, in conjunction with the presence of a history of contact with infected crabs or other individuals. Laboratory tests may be performed to detect the presence of the Pincerovirus in bodily fluids or respiratory samples to confirm the diagnosis.

Currently, there is no specific treatment or cure for pincer pox. Therefore, the focus of management primarily centers on ensuring the safety and well-being of individuals experiencing the crab-like raving behavior. It is crucial to monitor these individuals closely and provide a safe rave environment to prevent them from inadvertently hurting themselves during the rave.

Restraint should be avoided, as it may exacerbate agitation and distress. Instead, caregivers and healthcare professionals should employ supportive strategies to create a safe space. This may involve removing potentially hazardous objects from the immediate vicinity, cushioning hard surfaces, and ensuring there are no sharp edges or corners that could cause injuries. Affected individuals should be kept away from music as it can exacerbate the rave-like symptoms.

In addition to the physical environment, emotional support is essential. Calm and reassuring communication can help alleviate anxiety and promote a sense of security. Providing comforting objects, such as soft toys or blankets, may also help in creating a soothing atmosphere.

During the management of pincer pox, it is advisable to consult with healthcare professionals who can provide guidance and address any concerns. They can offer specific recommendations based on the individual's unique needs and ensure appropriate care and support are provided throughout the duration of the crab-like raving episode.

A.4 Prevention and Control

Preventing the spread of pincer pox requires a combination of measures. In crab-human transmission, individuals should exercise caution when handling crabs, especially those displaying unusual behavior or signs of illness. Wearing protective gloves and minimizing direct contact can reduce the risk of exposure.

In human-to-human transmission, practicing good respiratory hygiene, such as covering the mouth and nose while coughing or sneezing, is essential to prevent the release of virus-laden droplets into the air. Maintaining proper hand hygiene and avoiding close contact with infected individuals also play crucial roles in containing the disease.

Furthermore, public health authorities may implement measures such as contact tracing, isolation of infected individuals, and public awareness campaigns to minimize the spread of pincer pox within communities.