পারফরম্যান্স আনলক করা: Dart-এ কনকারেন্সি এবং প্যারালালিজম

এতক্ষণে আপনারা হয়তো থ্রেড (thread) সম্পর্কে শুনেছেন এবং কীভাবে এটি অ্যাপকে একই সাথে একাধিক কাজ করতে সাহায্য করে, সবকিছু মসৃণ রাখে। Dart-এর ক্ষেত্রে ব্যাপারগুলো একটু ভিন্ন। যদিও আমরা অন্যান্য ভাষার মতো প্রথাগত থ্রেড নিয়ে তেমন একটা কাজ করি না, Dart কনকারেন্সি (concurrency) (একাধিক কাজ একই মাইক্রোসেকেন্ডে না করেও সেগুলোর অগ্রগতি করা) এবং প্যারালালিজম (parallelism) (প্রসেসরের বিভিন্ন কোরে একই সাথে একাধিক কাজ করা) উভয়ের জন্যই শক্তিশালী ম্যাকানিজম সরবরাহ করে।

এই ধারণাগুলো *কী* বোঝায় তা আমরা ইতিমধ্যে আলোচনা করেছি। এখন, আসুন দেখি *কেন, কীভাবে এবং কখন* আপনি আপনার প্রিয় Dart অ্যাপ্লিকেশনগুলোতে আরও প্রতিক্রিয়াশীল এবং পারফরম্যান্ট সফ্টওয়্যার তৈরি করতে এগুলো ব্যবহার করবেন।

কেন এতকিছু? গতি এবং প্রতিক্রিয়াশীলতার প্রয়োজনীয়তা

ভাবুন আপনার Dart অ্যাপটিকে নেটওয়ার্ক থেকে ডেটা আনতে হবে, একটি বড় ফাইল পড়তে হবে বা সত্যিই জটিল কোনো গণনা করতে হবে। যদি আপনি এটি সরাসরি মেইন আইসোলেটে (main isolate) (যে একক থ্রেডে Dart অ্যাপ্লিকেশন শুরু হয়) করেন, আপনার পুরো অ্যাপ্লিকেশনটি জমে যাবে। কোনো ইউজার ইন্টারফেস (user interface) আপডেট হবে না, ক্লিকে কোনো সাড়া দেবে না – শুধু একটি হতাশাজনক বিরতি। এখানেই Dart-এর অ্যাসিঙ্ক্রোনাস (asynchronous) টুলগুলো কাজে আসে।

- **Concurrency (`async`/`await`)** I/O-বাউন্ড অপারেশনের জন্য অসাধারণ। যেমন নেটওয়ার্ক রিকোয়েস্ট, ফাইল সিস্টেম অ্যাক্সেস বা টাইমারের জন্য অপেক্ষা করা। আপনার অ্যাপ একটি অপারেশন শুরু করতে পারে, তারপর সেটি সম্পন্ন হওয়ার জন্য অপেক্ষা করার সময় অন্য কিছু করতে পারে, যা UI-কে জমে যাওয়া থেকে রক্ষা করে। এটা অনেকটা একজন শেফের পানি গরম করতে (একটি I/O টাস্ক) দিয়ে তারপর সবজি কাটার মতো, যখন পানি গরম হচ্ছে। তারা একাধিক কাজ একই সময়ে করছে, কিন্তু দুটোই একই মুহূর্তে না-ও হতে পারে।
- **Parallelism (Isolates)** আপনার CPU-বাউন্ড অপারেশনের জন্য সেরা। এই কাজগুলো সত্যিই সংখ্যা নিয়ে কাজ করে এবং প্রসেসর দখল করে রাখে, যেমন বিশাল JSON ফাইল পার্স করা, জটিল ইমেজ প্রসেসিং বা ক্রিপ্টোগ্রাফিক ক্যালকুলেশন। Isolate-গুলো Dart-কে অন্যান্য প্রসেসর কোরে কোড এক্সিকিউট করতে দেয়, সত্যিকার অর্থে প্যারালালি, আপনার প্রধান অ্যাপ্লিকেশনের প্রতিক্রিয়াশীলতা ব্লক না করে। এটা অনেকটা রান্নাঘরে একাধিক শেফ থাকার মতো, প্রত্যেকে একই সাথে বিভিন্ন জটিল ডিশ তৈরি করছে।

Dart কীভাবে এটি করে: ইভেন্ট লুপ এবং আইসোলেট (Event Loop and Isolates)

প্রতিটি আইসোলেটের মধ্যে Dart-এর প্রধান এক্সিকিউশন মডেলটি একক-থ্রেডেড, যা একটি **ইভেন্ট লুপের (event loop)** চারপাশে ঘোরে। যখন আপনি একটি আইসোলেটের মধ্যে `async` এবং `await` ব্যবহার করেন, তখন আপনি Dart-কে বলছেন, "এই অপারেশনটিতে কিছুটা সময় লাগতে পারে। শুধু বসে থেকো না; অন্য ইভেন্টগুলো প্রসেস করো এবং এই `Future` সম্পন্ন হলে এখানে ফিরে এসো।" এটি I/O টাস্কগুলোর জন্য দুর্দান্ত কারণ অপেক্ষার সময় CPU আসলে ব্যস্ত থাকে না; এটি শুধু নেটওয়ার্ক বা ডিস্ক থেকে ডেটা আসার জন্য অপেক্ষা করে।

কিন্তু যদি CPU নিজেই বটেলনেক (bottleneck) হয়? এখানেই **Isolates** কাজে আসে।

একটি Isolate হলো একটি স্বাধীন কর্মীর মতো যার নিজস্ব মেমরি এবং নিজস্ব ইভেন্ট লুপ রয়েছে। **গুরুত্বপূর্ণভাবে, আইসোলেটগুলো আপনার প্রধান অ্যাপ্লিকেশনের আইসোলেটের (বা অন্য কোনো আইসোলেটের) সাথে মেমরি শেয়ার করে না।** প্রতিটি আইসোলেটের নিজস্ব মেমরি হিপ (memory heap) থাকে, যা নিশ্চিত করে যে শেয়ার্ড ডেটার উপর কোনো রেস কন্ডিশন (race condition) নেই এবং প্রথাগত মাল্টিথ্রেডিং-এ পাওয়া জটিল লকিং ম্যাকানিজমের প্রয়োজনীয়তা এড়িয়ে যায়। এটি Dart-এর কনকারেন্সি মডেলের একটি মৌলিক দিক, যা নিরাপত্তা এবং সরলতার জন্য ডিজাইন করা হয়েছে।

আইসোলেটগুলোর মধ্যে যোগাযোগ: মেসেজ পাসিং, `SendPort`, এবং `ReceivePort`

যেহেতু আইসোলেটগুলো মেমরি শেয়ার করে না, তাদের মধ্যে যোগাযোগ সম্পূর্ণভাবে **মেসেজ পাসিংয়ের (passing messages)** মাধ্যমে ঘটে। এটিকে বিভিন্ন অফিসের মধ্যে চিঠি বা পার্সেল আদান-প্রদানের মতো ভাবুন—এটি নিরাপদ, নিয়ন্ত্রিত এবং অ্যাসিঙ্ক্রোনাস।

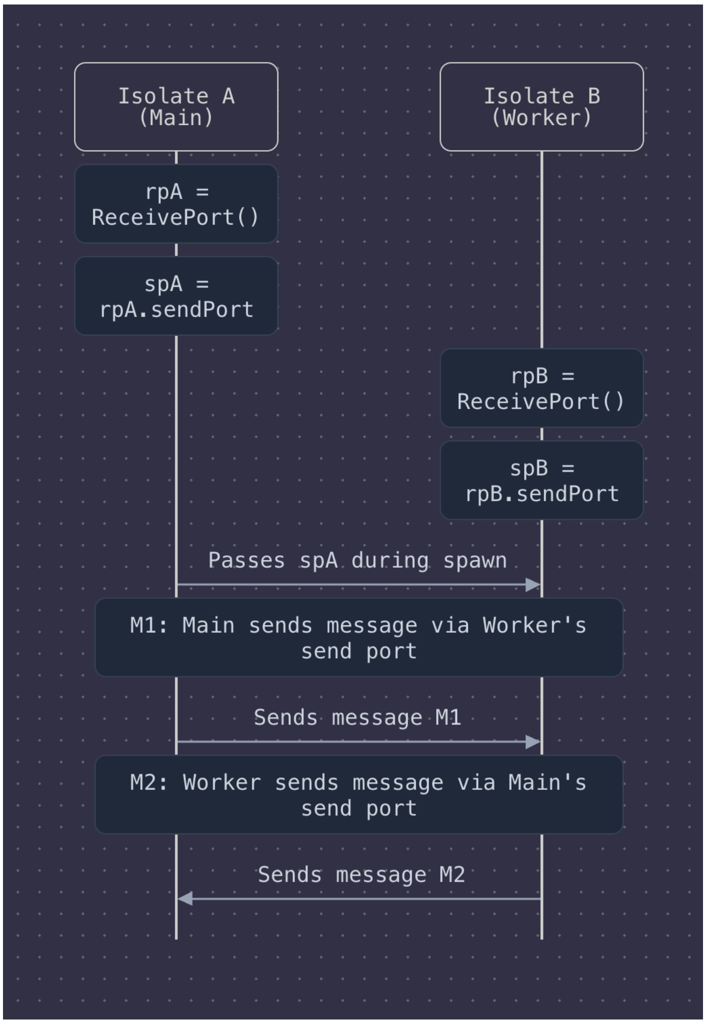
মেসেজ পাসিং আসলে কীভাবে কাজ করে: যখন একটি আইসোলেট অন্যটিতে একটি মেসেজ পাঠায়, তখন Dart রানটাইম কার্যকরভাবে মেসেজটি কপি করে (বা নির্দিষ্ট ধরণের ডেটার মালিকানা স্থানান্তর করে) প্রেরকের মেমরি হিপ থেকে প্রাপকের মেমরি হিপে। এটি নিশ্চিত করে যে আইসোলেশন বজায় থাকে। গ্রহণকারী আইসোলেট তারপর তার ইভেন্ট লুপের মাধ্যমে এই মেসেজটি প্রসেস করে।

এই প্রক্রিয়াটি `SendPort` এবং `ReceivePort` অবজেক্ট দ্বারা সহজতর হয়:

- `ReceivePort`: একটি আইসোলেট আগত মেসেজ শোনার জন্য একটি `ReceivePort` তৈরি করে। এটি একটি `Stream` এক্সপোজ করে যা মেসেজ আসার সাথে সাথে তা নির্গত করে। একটি `ReceivePort`-এর একটি সংশ্লিষ্ট `SendPort` থাকে।
- `SendPort`: `SendPort` হলো তার `ReceivePort`-এর জন্য মেইলিং ঠিকানার মতো। আপনি অন্য আইসোলেটে একটি `SendPort` পাস করতে পারেন, যার ফলে সেই আইসোলেটটি তার সাথে যুক্ত `ReceivePort`-এ মেসেজ ফেরত পাঠাতে পারে।

`SendPort` এবং `ReceivePort`-এর দৃশ্যায়ন:

দুটি আইসোলেট কল্পনা করুন, Isolate A (Main) এবং Isolate B (Worker):



১. **সেটআপ (Setup)**: * Isolate A তৈরি করে `rpA = ReceivePort()` এবং এর `spA = rpA.sendPort` পায়। * Isolate A যখন Isolate B স্পন (spawn) করে, তখন এটি প্রাথমিক মেসেজের অংশ হিসাবে `spA` কে Isolate B-তে পাস করতে পারে। * Isolate B যদি সরাসরি A থেকে (শুধু প্রাথমিক সেটআপ মেসেজ ছাড়াও) বা অন্যান্য আইসোলেট থেকে মেসেজ গ্রহণ করতে চায়, তাহলে এটি নিজস্ব `rpB` এবং `spB` তৈরি করতে পারে। সাধারণ রিকোয়েস্ট-রেসপন্সের জন্য, Isolate A তার `spA` কে Isolate B-তে পাস করতে পারে এবং Isolate B ফলাফল ফেরত পাঠানোর জন্য এই `spA` ব্যবহার করে।

২. **মেসেজ পাঠানো (Sending a Message)**: * Isolate A, Isolate B-তে ডেটা পাঠাতে চায়। এটি একটি `SendPort` ব্যবহার করে যা Isolate B এক্সপোজ করেছে (যেমন, `workerSendPort` যা B স্পন করার সময় প্রাপ্ত হয়েছিল বা B থেকে ফেরত পাঠানো হয়েছিল)। * `workerSendPort.send(messageData);`

৩. **মেসেজ গ্রহণ করা (Receiving a Message)**: * Isolate B-এর `ReceivePort` (`rpB`) তার স্ট্রিমের মাধ্যমে `messageData` নির্গত করবে। * Isolate B এই স্ট্রীমটি শোনে: `rpB.listen((dynamic message) { /* process message */ });`

কী ধরনের ডেটা পাঠানো যায়? মেসেজগুলো সাধারণত কপি করা হয়, তাই ডেটা অবশ্যই "প্রেরণযোগ্য" (sendable) হতে হবে। এর মধ্যে রয়েছে:

- প্রিমিটিভ টাইপ (Primitive types): `null`, `bool`, `int`, `double`, `String`।
- `SendPort` এবং `Capability`-এর ইনস্ট্যান্স।
- List এবং Map যার উপাদানগুলো নিজেরাই প্রেরণযোগ্য টাইপ।
- কিছু বিশেষ অবজেক্ট যা কপি না করে দক্ষতার সাথে স্থানান্তর করা যায়, যেমন `Uint8List` (transferable objects)।

সীমাবদ্ধতা (Limitations):

- **জটিল অবজেক্ট (Complex Objects)**: নির্বিচারে ক্লাসের ইনস্ট্যান্স যা সম্পূর্ণরূপে প্রেরণযোগ্য টাইপ দ্বারা গঠিত নয়, সেগুলো সরাসরি প্রেরণযোগ্য নাও হতে পারে যদি সেগুলোতে উদাহরণস্বরূপ, নেটিভ রিসোর্স বা অন্যান্য অ-স্থানান্তরযোগ্য স্টেট থাকে। আপনাকে সাধারণত সেগুলোকে একটি প্রেরণযোগ্য ফরম্যাটে (যেমন, JSON) সিরিয়ালাইজ করতে হবে এবং তারপরে গ্রহণকারী আইসোলেটে ডিসিরিয়ালাইজ করতে হবে।
- **ক্লোজার (Closures - Anonymous Functions)**: আপনি সরাসরি নির্বিচারে ক্লোজার মেসেজ হিসাবে পাঠাতে পারবেন না। `Isolate.spawn()` বা `compute()`-তে পাস করা ফাংশনগুলো অবশ্যই টপ-লেভেল ফাংশন বা স্ট্যাটিক মেথড হতে হবে। এর কারণ হলো একটি ক্লোজার তার চারপাশের লেক্সিক্যাল স্কোপ (পরিবেশ থেকে ভেরিয়েবল যেখানে এটি তৈরি হয়েছিল) ক্যাপচার করতে পারে এবং সেই স্কোপ নতুন আইসোলেটের মেমরিতে বিদ্যমান নেই।

আইসোলেট স্পন করা: `Isolate.spawn()` দিয়ে ম্যানুয়াল নিয়ন্ত্রণ এবং একটি সহায়ক: `compute()`

আপনি `Isolate.spawn()` ব্যবহার করে ম্যানুয়ালি আইসোলেট স্পন করতে পারেন, যা আপনাকে `ReceivePort` এবং `SendPort` এর সাথে দ্বিমুখী যোগাযোগ স্থাপনের উপর সূক্ষ্ম নিয়ন্ত্রণ দেয়।

```
import 'dart:isolate';

// নতুন আইসোলেটের জন্য এন্ট্রি পয়েন্ট
void _newIsolateEntry(SendPort mainSendPort) {
  ReceivePort newIsolateReceivePort = ReceivePort();
  mainSendPort.send(newIsolateReceivePort.sendPort); // এর SendPort প্রধান
আইসোলেটে পাঠান

```

```
  newIsolateReceivePort.listen((dynamic message) {
    if (message is String) {
      print('New Isolate received: $message');
      final result = message.toUpperCase() + " (processed by isolate)";
      mainSendPort.send(result);
    }
  });
}

// প্রধান আইসোলেটে
Future<void> useManualSpawn() async {
  ReceivePort mainReceivePort = ReceivePort();

  print('Main Isolate: Spawning new isolate.');
  Isolate newIsolate = await Isolate.spawn(_newIsolateEntry, mainReceivePort.sendPort);

  SendPort? newIsolateSendPort;
  Completer<void> isolateReadyCompleter = Completer();

  mainReceivePort.listen((dynamic message) {
    if (message is SendPort) {
      newIsolateSendPort = message;
      print('Main Isolate: Received SendPort from new isolate.');
      newIsolateSendPort?.send('Hello from Main Isolate!');
    } else if (message is String) {
      print('Main Isolate: Received result: $message');
      mainReceivePort.close(); // কাজ শেষে পোর্ট বন্ধ করুন
      newIsolate.kill(priority: Isolate.immediate); // আইসোলেট পরিষ্কার করুন
      if (!isolateReadyCompleter.isCompleted) {
        isolateReadyCompleter.complete();
      }
    }
  });
  await isolateReadyCompleter.future; // কমিউনিকেশন চক্র সম্পন্ন হওয়ার জন্য অপেক্ষা করুন
  print('Main Isolate: Manual spawn example finished.');
}
```

তবে, অনেক সাধারণ ব্যবহারের ক্ষেত্রে, **Flutter framework** একটি সহজ উচ্চ-স্তরের ফাংশন সরবরাহ করে: `compute()`। *(স্পষ্টীকরণ: `compute()` `flutter/foundation.dart` লাইব্রেরির অংশ এবং এটি Flutter অ্যাপ্লিকেশনগুলোর মধ্যে নির্বিঘ্নে কাজ করার জন্য ডিজাইন করা হয়েছে। Flutter ব্যবহার না করা পিওর Dart অ্যাপ্লিকেশনগুলোর জন্য, আপনি সাধারণত সরাসরি `Isolate.spawn()` ব্যবহার করবেন বা অনুরূপ কোনো সহায়ক প্রয়োগ করবেন।)*

আপনি যদি একটি Flutter পরিবেশে থাকেন, `compute()` একটি নতুন আইসোলেটে একটি ফাংশন চালানো সহজ করে তোলে:

```
// একটি Flutter অ্যাপে ( নিশ্চিত করুন যে আপনি flutter/foundation.dart ইম্পোর্ট করেছেন)
// import 'package:flutter/foundation.dart';
```

```
// এই ফাংশনটি একটি পৃথক আইসোলেটে কার্যকর হবে।
// এটি অবশ্যই একটি টপ-লেভেল ফাংশন বা একটি স্ট্যাটিক মেথড হতে হবে।
Map<String, dynamic> _parseJsonInIsolate(String jsonData) {
  print("Parsing JSON in a separate isolate...");
  // কিছু ভারী পার্সিং কাজের অনুকরণ করুন
  for (int i = 0; i < 1000000000; i++) {
    // শুধু কাজ অনুকরণ করার জন্য একটি ব্যস্ত লুপ
  }
  // একটি বাস্তব পরিস্থিতিতে, আপনি dart:convert ব্যবহার করতেন
  // final Map<String, dynamic> data = jsonDecode(jsonData);
  final Map<String, dynamic> data = {"message": "Parsed: $jsonData"}; // সরলীকৃত
  print("Parsing complete in isolate.");
  return data;
}

// আপনার Flutter উইজেট বা সার্ভিসে:
// await compute(_parseJsonInIsolate, massiveJsonString);
```

`Isolate.spawn()` বা `compute()`-তে পাস করা ফাংশন সম্পর্কে মূল বিষয়গুলো:

- এগুলো অবশ্যই **টপ-লেভেল ফাংশন বা স্ট্যাটিক মেথড** হতে হবে। ইনস্ট্যান্স মেথড বা ক্লোজার যা `this` বা লোকাল ভেরিয়েবল ক্যাপচার করে সেগুলো সরাসরি নতুন আইসোলেটের এন্ট্রি পয়েন্ট হিসাবে ব্যবহার করা যাবে না।

ওভারহেড এবং কখন আইসোলেট ব্যবহার করবেন (Overhead and When to Use Isolates)

একটি আইসোলেট স্পন করা বিনামূল্যে নয়। এর মধ্যে রয়েছে:

- নতুন আইসোলেটের জন্য একটি পৃথক মেমরি হিপ বরাদ্দ করা।
- সেই আইসোলেটের জন্য একটি নতুন ইভেন্ট লুপ শুরু করা।
- আইসোলেটগুলোর মধ্যে মেসেজ পাসিংয়ের খরচ (ডেটা কপি করা)।

কখন এগুলো ব্যবহার করা সার্থক?

- **সত্যিকারের ভারী CPU-বাউন্ড কাজ**: যদি একটি টাস্কে উল্লেখযোগ্য গণনা জড়িত থাকে যা শত শত মিলিসেকেন্ড বা এমনকি সেকেন্ড সময় নিতে পারে, প্রধান আইসোলেট ব্লক করে এবং UI জ্যাঙ্ক বা প্রতিক্রিয়াহীনতার কারণ হয়, তবে এটিকে অন্য আইসোলেটে অফলোড করা উপকারী। উদাহরণস্বরূপ জটিল ডেটা প্রসেসিং, ক্রিপ্টোগ্রাফি বা নিবিড় গণনা।
- **ছোট কাজের জন্য খরচ সুবিধার চেয়ে বেশি**: খুব সংক্ষিপ্ত গণনার জন্য (যেমন, কয়েক মিলিসেকেন্ড), একটি আইসোলেট তৈরি এবং মেসেজ পাস করার ওভারহেড প্যারালাল এক্সিকিউশনের মাধ্যমে সাশ্রয় হওয়া সময়ের চেয়ে বেশি হতে পারে। এই ধরনের ক্ষেত্রে, প্রধান আইসোলেটে কাজটি করা (যদি এটি প্রতিক্রিয়াশীলতাকে প্রভাবিত না করে) বা অ্যালগরিদম অপ্টিমাইজ করা ভাল হতে পারে।
- **I/O-এর জন্য `async`/`await` ব্যবহার করুন**: I/O-বাউন্ড অপারেশনের জন্য (নেটওয়ার্ক রিকোয়েস্ট, ফাইল অ্যাক্সেস), যেখানে প্রোগ্রাম তার বেশিরভাগ সময় *অপেক্ষা* করে কাটায়, প্রধান আইসোলেটে `async`/`await` সাধারণত যথেষ্ট এবং নতুন আইসোলেট স্পন করার চেয়ে বেশি হালকা। একটি আইসোলেট একটি I/O অপারেশন দ্রুত সম্পন্ন করবে না যদি বটেলনেক বাহ্যিক সিস্টেম বা নেটওয়ার্ক হয়।

সংক্ষেপে: ওভারহেডকে ন্যায্যতা দেওয়ার জন্য যথেষ্ট কম্পিউটেশনালি নিবিড় কাজগুলোর জন্য বিচক্ষণতার সাথে আইসোলেট ব্যবহার করুন।

আইসোলেটস: কনকারেন্সি না প্যারালালিজম? (Isolates: Concurrency or Parallelism?)

এটি একটি দারুণ প্রশ্ন! Dart-এর আইসোলেটগুলো **কনকারেন্সি এবং সত্যিকারের প্যারালালিজম উভয়ই** অর্জন করতে পারে।

- **কনকারেন্সি (Concurrency)**: এমনকি একটি সিঙ্গেল-কোর প্রসেসরেও আপনার একাধিক আইসোলেট থাকতে পারে। যদিও সেই সিঙ্গেল কোরে যেকোনো একটি মাইক্রোসেকেন্ডে শুধুমাত্র একটি আইসোলেট চলতে পারে, অপারেটিং সিস্টেম তাদের মধ্যে স্যুইচ করতে পারে, যার ফলে তারা তাদের কাজগুলোতে *কনকারেন্টলি* অগ্রগতি করতে পারে। প্রধান আইসোলেট প্রতিক্রিয়াশীল থাকতে পারে যখন অন্য একটি আইসোলেট পটভূমিতে কাজ করে।
- **প্যারালালিজম (Parallelism)**: যদি আপনার ডিভাইসে একাধিক CPU কোর থাকে, Dart বিভিন্ন কোরে একই সাথে বিভিন্ন আইসোলেট চালাতে পারে (এবং সাধারণত চালায়)। এটি **সত্যিকারের প্যারালালিজম**, যেখানে একাধিক কোড লাইন একই সময়ে এক্সিকিউট হচ্ছে।

সুতরাং, আইসোলেটগুলো ডিফল্টরূপে কনকারেন্সির জন্য একটি ম্যাকানিজম সরবরাহ করে (স্বাধীন অগ্রগতির অনুমতি দেয়), এবং হার্ডওয়্যার রিসোর্স (একাধিক কোর) উপলব্ধ থাকলে তারা প্যারালালিজম সক্ষম করে। CPU-বাউন্ড কাজের জন্য আইসোলেট ব্যবহারের প্রাথমিক লক্ষ্য হলো এই প্যারালালিজম অর্জন করা এবং প্রধান আইসোলেট (প্রায়শই UI-এর জন্য দায়ী) জমে যাওয়া থেকে রোধ করা।

বাস্তব জীবনের ব্যবহার: একটি সার্ভার অ্যাপ্লিকেশনের জন্য ব্যাকগ্রাউন্ড ডেটা প্রসেসিং

(আপনার মূল ডকুমেন্ট থেকে এই বিভাগটি মূলত প্রাসঙ্গিক, তবে নিশ্চিত করুন যে `Isolate.compute`কে Flutter-নির্দিষ্ট হিসাবে স্পষ্ট করা হয়েছে বা একটি পিওর Dart উদাহরণের জন্য `Isolate.spawn` ব্যবহার করুন।)

একটি পিওর Dart সার্ভার অ্যাপ্লিকেশন বিবেচনা করা যাক। যদি এটিকে বড় লগ ফাইল প্রসেস করতে হয়:

আইসোলেট দিয়ে সমাধান (পিওর Dart-এর জন্য `Isolate.spawn` ব্যবহার করে):

১. মেইন আইসোলেট একটি রিকোয়েস্ট গ্রহণ করে। ২. এটি `Isolate.spawn()` ব্যবহার করে প্রসেসিংকে একটি নতুন আইসোলেটে পাঠায়। ফলাফল ফেরত পাওয়ার জন্য এটির `ReceivePort` সেট আপ করতে হবে। ৩. নতুন আইসোলেট ফাইলটি প্রসেস করে (CPU-intensive)। এটি উপলব্ধ থাকলে একটি ভিন্ন কোরে ঘটে। ৪. মেইন আইসোলেট অন্যান্য রিকোয়েস্টের প্রতি প্রতিক্রিয়াশীল থাকে। ৫. ওয়ার্কার আইসোলেট তার প্রাপ্ত `SendPort`-এর মাধ্যমে ফলাফলটি মেইন আইসোলেটে ফেরত পাঠায়। ৬. মেইন আইসোলেট ফলাফলটি পরিচালনা করে।

ধারণাগত কোড স্নিপেট (পিওর Dart সাথে `Isolate.spawn`):

```
import 'dart:isolate';
import 'dart:io'; // ফাইল অপারেশনের জন্য
import 'dart:async'; // Completer-এর জন্য

// এই ফাংশনটি নতুন আইসোলেটে চলে
Future<void> _processLogFileIsolateEntry(Map<String, dynamic> message)
async {
  SendPort replyPort = message['replyPort'] as SendPort;
  String filePath = message['filePath'] as String;

  print("[Isolate ${Isolate.current.debugName}] Processing log file:
```

```
$filePath");
  String report = "Report for $filePath:\n";
  try {
    // ফাইল পড়া এবং CPU-নিবিড় পার্সিং ও বিশ্লেষণের অনুকরণ
    // String fileContent = await File(filePath).readAsString(); // উদাহরণ:
ফাইল পড়া
    // report += "Characters: ${fileContent.length}\n"; // উদাহরণ: বিশ্লেষণের
অংশ

    // ভারী কাজের অনুকরণ
    for(int i=0; i < 2000000000; i++){ /* busy work */ }
    report += "Analysis complete. Found 5 critical errors.\n";
    print("[Isolate ${Isolate.current.debugName}] Finished processing
$filePath");
    replyPort.send({'status': 'success', 'report': report});
  } catch (e, s) {
    print("[Isolate ${Isolate.current.debugName}] Error processing
$filePath: $e");
    replyPort.send({'status': 'error', 'error': e.toString(),
'stackTrace': s.toString()});
  }
}

Future<String> processLogFileWithIsolate(String filePath) async {
  print("[Main Isolate] Received request to process log file: $filePath");
  final p = ReceivePort();
  final completer = Completer<String>();

  try {
    // আইসোলেট স্পন করুন
    Isolate isolate = await Isolate.spawn(
      _processLogFileIsolateEntry,
      {'replyPort': p.sendPort, 'filePath': filePath},
      onError: p.sendPort, // পোর্টে ত্রুটিও পাঠান
      onExit: p.sendPort    // প্রস্থান বার্তা পাঠান
    );

    p.listen((dynamic message) {
      if (message is Map && message['status'] == 'success') {
        completer.complete(message['report'] as String);
        p.close();
        isolate.kill(priority: Isolate.immediate);
      } else if (message is Map && message['status'] == 'error') {
        completer.completeError(
          Exception("Isolate error: ${message['error']}"),
          StackTrace.fromString(message['stackTrace'] as String? ?? '')
        );
        p.close();
        isolate.kill(priority: Isolate.immediate);
      } else if (message == null) { // onExit message
        if (!completer.isCompleted) {
            completer.completeError(Exception("Isolate exited unexpectedly
before sending a result."));
        }
```

```
        p.close();
      }
    });
  } catch (e,s) {
      completer.completeError(e,s);
      p.close();
  }
  return completer.future;
}

// উদাহরণ ব্যবহার:
// try {
//   String report = await processLogFileWithIsolate("path/to/user_uploaded_log.txt");
//   print("[Main Isolate] Log file processing complete. Report:\n$report");
// } catch (e) {
//   print("[Main Isolate] Error processing log file: $e");
// }
```

উপসংহার (Wrapping Up)

কখন এবং কীভাবে `async`/`await` এর সাথে কনকারেন্সি এবং Isolates এর সাথে প্যারালালিজম ব্যবহার করতে হয় তা বোঝা উচ্চ-পারফরম্যান্স, প্রতিক্রিয়াশীল Dart অ্যাপ্লিকেশন লেখার মূল চাবিকাঠি।

- `async`/`await` আপনার অ্যাপকে অপেক্ষার সময় জমে যাওয়া থেকে রক্ষা করে, I/O-বাউন্ড কাজের জন্য আদর্শ।
- **Isolates** আপনাকে ভারী CPU-বাউন্ড কাজের জন্য মাল্টি-কোর প্রসেসরের সম্পূর্ণ শক্তি ব্যবহার করতে দেয়, সত্যিকারের প্যারালালিজম অর্জন করে।
  - তারা পৃথক মেমরি হিপের কারণে **স্পনিং আইসোলেট থেকে ভেরিয়েবল অ্যাক্সেস করতে পারে না**।
  - `Isolate.spawn()` (বা Flutter-এর `compute()`) এ পাস করা ফাংশনগুলো অবশ্যই **টপ-লেভেল বা স্ট্যাটিক** হতে হবে।
  - আইসোলেট **শুধুমাত্র সত্যিকারের ভারী কাজের জন্য ব্যবহার করুন**; অন্যথায়, ওভারহেড (স্পনিং খরচ, মেসেজ পাসিং) সুবিধার চেয়ে বেশি হতে পারে।

পরীক্ষা শুরু করুন! আপনার Dart প্রকল্পগুলোতে I/O-বাউন্ড এবং CPU-বাউন্ড বাধাগুলো চিহ্নিত করুন এবং দেখুন এই সরঞ্জামগুলো কীভাবে পার্থক্য তৈরি করতে পারে। হ্যাপি কোডিং!