International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I5P127

Volume 72 Issue 5, 217-222, May 2024
© 2024 Seventh Sense Research Group®

Original Article

Optimization with Interprocess Communication via Channels - A CSP Approach

Priyanka Nawalramka

Staff Software Engineer, House Canary, Boulder, CO, USA.

Corresponding Author : priyanka.nawalramka@gmail.com

Received: 03 April 2024 Revised: 05 May 2024 Accepted: 15 May 2024 Published: 27 May 2024

Abstract *- In production software development, when handling large-scale data, performance becomes an essential consideration. Given the rate at which data is generated and consumed today in modern cloud based systems, it, in fact, becomes an essential requirement. The choice of tools, algorithm design and programming patterns can all help in gaining as much optimization as possible within an application. Concurrency plays a vital role in software and resource optimization. Traditional concurrency models use locks with multithreading over shared memory as the synchronization primitive. The Communicating Sequential Processes (CSP) model uses communication as the synchronization primitive. This article delves briefly into the theory of CSP. This is followed by a discussion on how to query and process big chunks of data from a database in an optimal manner using interprocess communication. The article also describes a case study using channels, an interprocess communication technique that uses message passing (based on CSP) with the concurrency constructs of the Go programming language.*

Keywords *- Channels, Communicating sequential processes, Concurrency, Data processing, Interprocess communication, Optimization.*

1. Introduction

In computer science, the desire for greater speed in executing tasks led to the introduction of parallelism [1]. Concurrency is the useful mechanism of dividing a task into smaller tasks that can be run independently and allowing them to progress at practically the same time. Concurrency is not the same as pure parallelism, where the different tasks execute simultaneously (typically in a multiprocessor system). An important feature of concurrent program execution is that it gives rise to indeterminacy in the behavior and outcome of a program.

The traditional concurrency model relies upon synchronization via locks (mutex, semaphore, condition variable, monitor, etc.) over a shared memory location. In 1978, C. A. R. Hoare introduced a model and formal language for using communication as the synchronization primitive.

Decades later today, memory based synchronization remains the most popular choice in programming. Nonetheless, the communication-based concurrency model did find adoption in some well-known programming languages, including Go, a newer language developed in the last two decades. The language creators emphasize the linguistic simplicity of the notation.

In the following sections, the concepts and notations of CSP are discussed in brief. Then, a case study is presented, including pseud code, which describes optimization methods for a practical and common scenario of querying bulk data from a database and processing them using interprocess communication techniques based on the previously referenced communication-based concurrency methodology.

2. Communicating Sequential Processes

Communicating Sequential Processes is a programming language model first introduced by C. A. R. Hoare in 1978, which described a parallel composition of communicating sequential processing as a fundamental program structuring method using input and output as basic primitives of programming [1]. The paper described a model using communication via strictly synchronous message passing as the basic concurrency primitive and provided recipes for solutions to several common programming problems, like bounded buffer, dining philosophers, etc., using the proposed model. In this version of CSP, messages were exchanged between specific process identities versus any particular middle entity, like a port number or a channel. The notation for an input command was defined as:

$$< source_process >? < target\ value >$$

This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

Whereas an output command was defined as:

$$< destination_process > ! < value >$$

This initial concurrent programming model led to the development of a formal mathematical model in the following years. CSP, as known today, refers to this mathematical model. The model uses process algebra to define semantics for various processes and their interaction with each other and with their environment. It describes methods to consider indeterminacy within a concurrent system. The model's core principle lies in the fact that a sequential composition of instructions can correctly represent the parallel composition of subsystems. It conceptualizes the idea of decomposing a computer system into subsystems running concurrently and constantly interacting with one another and, more importantly, with their common environment [1].

2.1. Primitives and Operators

CSP defines an event to be the fundamental unit of a process, and that all events are regarded as instantaneous and indivisible. Furthermore, the behavior of any process up to some moment in time can be recorded as a trace (the sequence of all events that happened during and leading up to that moment in time). The behavior and transitions of the processes are described using these primitives and algebraic operators. Operators are defined to represent interleaving (completely independent) processes and hiding of events (abstraction via making an event unobservable), among others. A fundamental notation of CSP defines a system engaging in event *e* and then behaving like process *P* using a prefix:

$$e -> P \ (e \ then \ P)$$

A deterministic choice allows for a process to evolve into one or the other form based on an initial event. The external environment is allowed to resolve the choice by selecting the initial event.

$$P \Box Q$$

A nondeterministic choice is one in which the external environment has no control over. Thus, abstraction within a system can be achieved by deciding to "ignore or conceal" the nondeterministic behavior non-relevant to the user. [1]

$$P \sqcap Q$$

The communication between processes is described as happening via a channel (synchronized) using message passing, with buffering as an option.

2.2. Limitations

In his original paper [1], Hoare recognized the fact that in the CSP model, processes have the possibility of ending up in a deadlock state. This would happen when a group of processes that are attempting to communicate with each other never actually correspond, in effect being in a wait-forever state.

Due to its performance overhead, CSP was only adopted in a few programming languages early on, and the memory-based concurrency models remained the prime method for obtaining concurrency.

3. CSP implementation in Programming Languages

The CSP model discussed was adopted early in programming languages such as Ocamm and Erlang. Some programming languages like Go (a modern programming language developed at Google) provide higher level communication-based concurrency primitives in addition to the traditional memory based synchronization constructs. Following the basic theory of CSP, which allows for the "parallel composition of subsystems" to be correctly represented via a "sequential composition of instructions" in a system wherein the components interact with one another - the result of a data race free Go program is deemed similar to that of a system where all the "goroutines are multiplexed onto a single processor" and hence run sequentially, accounting for the interactions between them. This behavior is technically termed as "DRF-SC (data race-free programs execute in a sequentially consistent manner)" [5].

The concurrency primitives of Go mainly include goroutines and channels. The shared memory based constructs like mutex and condition variable implementations are also present in the language as part of the sync package. However, the language creators prefer the use of channels over the other primitives.

3.1. Goroutines

Goroutines are green threads managed by the Go language runtime. The runtime manages the lifecycle and multiplexes the goroutines onto low-level operating system threads. Direct OS (Operating System) threads are not exposed by the language APIs (Application Programming Interface). The goroutines can interact with one another and establish a relative order for execution using various synchronization mechanisms such as a communication-based channel or memory based locks, like mutex.

3.2. Channels

Channels in Go are directly inspired by those mentioned above: Communicating Sequential Processes (CSP) formal language in computer science. It is an interprocess communication technique based on message passing. Messages can optionally be stored in a FIFO (First-In-First-Out) buffer, and sends are blocked in Go when the buffer is full. It follows the principle described in Hoare's original paper [1], which states that communicating input and output commands must be synchronized, and a delay must be

introduced when one of them is ready before the other. The delay ends when the other command either becomes ready to process the message or is terminated. Input and output commands in the previous statement are analogous to receiving and sending on a channel.

Multiple channels can be chained together to create a pipeline. Although Go's channel implementation does not implement the actor model in concurrent systems, it can be simulated using this construct if desired.

According to the official Go memory model reference: "the kth receive on a channel with capacity C is synchronized before the k+Cth send from that channel completes" [5]. It should be noted that this creates a perfect construct for implementing a counting semaphore using buffered channels in Go. The maximum and active semaphore counts can be represented by the buffer size and current message count in the channel, respectively. Sending to the channel indicates acquiring the semaphore, and receiving from the channel is analogous to releasing the semaphore. The same approach can be generalized to limit concurrency as well. The idea of a counting semaphore implementation using communication-based concurrency is also mentioned in Hoare's original paper on CSP [1].

With this brief introduction to concurrency concepts, the proceeding topics will cover the practical scenario of handling large scale data in a software system. The following discussion and examples demonstrate how to query and process data of the order of tens or even hundreds of thousands of records with efficient resource utilization using interprocess communication. In order to keep the discussion simple, all database interactions are in abstract form, while the main emphasis lies on how the various processes communicate with each other.

4. Case Study

An application relies on large amounts of data stored in a database table and needs to query and process the data on a daily basis. Examples may include a system that reads from a daily record of Amazon book reviews or social media feeds and performs sentiment analysis on each record. Loading the entire dataset into memory is not practical due to its size. Furthermore, the sentiment analysis process may involve interaction with an external system, which implies significant I/O (input/output) operations will occur.

5. Solution Brief

The problem of reading large datasets with limited memory can be solved by iterating or paginating over the dataset via a database cursor instead of a single load. Furthermore, the task of fetching the records from the database can be decoupled from the processing of those records. As records are fetched iteratively, they must be processed with as minimum resource usage as possible. This can be achieved via grouping the records into batches, separation of concerns by delegating processing logic to another process or subtask and carrying out some or all of the tasks concurrently. Lastly, all subtasks involved must communicate and share data in a synchronized manner.

6. Abstractions

All mainstream languages provide connector libraries to interact with major databases. Consider these API contracts in order to interact with a database. DBRowsCursor is an interface that spells out the API contracts in order to operate on the results from a database query.

```
type DBRowsCursor interface {
	Close()
	Next()
	Scan(any) error
	Err() error
}
```

Result is an abstract data type to encapsulate a single record of data inside a Go struct.

```
type Result struct { // contains fields mapping to database columns}
```

dbQuerier defines a function type, the implementation of which should allow for querying a database table and returning the resulting data, including any errors.

```
// the implementing function should know how to retrieve data from
a database
type dbQuerier func() (DBRowsCursor, error)
```

7. Fetching Data and Iterating Over the Result Set

It is not uncommon to see practical implementations in which a large request is made to the database, and the entire result set is loaded into memory for processing. This approach has limitations and can utilize too many resources if careful attention is not paid. It will simply not work when the result set in question is in the order of thousands or more. So, instead, it is appropriate to request the data one row at a time via a cursor. Consider a run query() pseudo function which accepts these arguments:

- querier - a function that knows how to query a database and returns a cursor to iterate over the results.
- resultC - a channel to which results will be sent.
- errC- a channel to which errors will be sent.

```
func runQuery(querier dbQuerier, resultC chan *Result, errC chan
error) {
	rows, err := querier()
	if err != nil {
		errC <- err
		return
	}
```

```
    defer rows.Close()

    for rows.Next() {
        r := Result{}
        if err := rows.Scan(&r); err != nil {
            errC <- err
    } else {
            resultC <- &r
        }
    }
    if err := rows.Err(); err != nil {
        errC <- err
    }
}
```

Several modern programming languages, including Go, support at least some form of functional programming paradigm. It can be achieved in Go using first-class functions. Therefore, the runQuery implementation is simplified by emphasizing result iteration by accepting the querier function as an argument. It should be noted that the driver implementation used to interact with the database must provide the cursor mechanism to advance over the result set iteratively. Results are sent over the resultC channel. Any errors encountered are sent over the errC channel.

8. Interprocess Communication

Figure 1 illustrates the interaction between the various processes within the application. The main thread drives the interaction by calling the corresponding function to fetch the data. It also creates goroutines and handles the processing of the data as well as errors. The result set is processed in batches for optimal performance. Finally, it ensures all tasks run to completion by synchronizing and waiting for their execution.

The processBigData() pseudo function performs the driver functionalities of:

- Delegating tasks by creating goroutines.
- Facilitating the interprocess communication.
- Handling data processing in batches and errors.

```
const batchSize = 100

func processBigData() int64 {
    batch := make([]*Result, 0, batchSize)
    // make a buffered channel to receive results in batches
    batchC := make(chan *Result, batchSize)
    errC := make(chan error)
    var wg sync.WaitGroup
    wg.Add(2)
…
}
```

Two different goroutines are instantiated within this function. Goroutine 1 listens for errors encountered over the errC channel. The err, ok := <-errC multi-value receives expression blocks until a message is available to consume. It returns any error in the err variable when available. ok will be set as false if the channel is closed, indicating no more messages can be received over this channel, thus breaking the infinite loop.

```
go func() {  // goroutine 1
        defer wg.Done()
        for {
            err, ok := <-errC
            if !ok {
                return
            }
            // handle the error
        }}()
```

Goroutine 2 listens over batchC, a buffered channel to receive database records as they are read. It maintains an in-memory fixed size buffer and sends off the result batch to process when it is full. Sends to a buffered channel are blocked when the buffer is full and receives are blocked when the buffer is empty (nothing to consume).

```
go func() {  // goroutine 2
        defer wg.Done()
        for record := range batchC {
        batch = append(batch, record)
        if len(batch) == batchSize {
            if err := processBatch(batch); err !=
nil { // handle the error}
            batch = batch[:0]
        }
}}()
```

Instead of batch processing, a fan-out approach can also be implemented by spinning off a worker goroutine.

As per the result received. The channel buffering would provide synchronization and safety against overutilization of resources by limiting the number of concurrent goroutines since sends are blocked when the channel buffer is full.

The data is retrieved by calling the runQuery() function.

```
querierFn := func() (DBRowsCursor, error) { /* implements
dbQuerier */ }
    runQuery(querierFn, batchC, errC)
```

Once runQuery() completes execution after reading the entire result set; the goroutines need to be signaled somehow that no more messages will be sent.

```
close(batchC)  // marker 1: signals goroutine 2 to exit
    close(errC)  // marker 2: signals goroutine 1 to exit
    wg.Wait() // marker 3: wait for goroutines to exit
gracefully
```

Closing a channel is analogous to sending a signal to a process in Go. At marker 1, channel batchC is closed in order to signal goroutine 2 to exit. At marker 2, channel errC is closed in order to signal goroutine 1 that no more messages are available to send, hence exit. At marker 3, the execution waits while the two goroutines complete the in-flight tasks and gracefully exit.

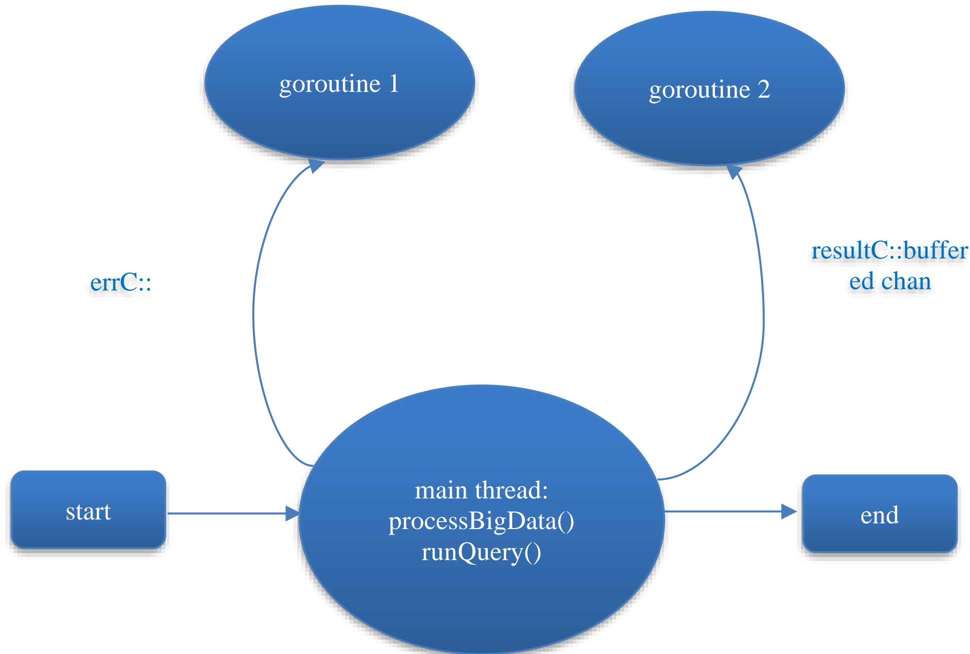

Fig. 1 Illustration of interprocess communication

Lastly, the last batch of unprocessed records is sent off for processing.

```
// process last batch if not empty
        if len(batch) > 0 {
                if err := processBatch(batch); err != nil {
                        // handle the error
                }}
```

9. Discussion

The case study described above addresses a common use case in modern software systems. Dealing with vast amounts of data has become increasingly popular, with applications moving to cloud infrastructure and faster computation with better hardware and new technologies. In light of these advancements, a system must be designed for optimal performance.

While applying distributed computing concepts and adding more hardware allows for scalability, applications must be designed with the goal of extracting optimal performance from resources within a single system first. This is where concurrency concepts are essential. A big task like extracting and processing large amounts of data should be split and organized into smaller subtasks. With the use of concurrency principles, many of these tasks can be progressed in parallel.

While the theory of CSP, introduced by Hoare, forms the basis of communication-based concurrency, it is crucial to know its limitations and the performance overhead involved. CSP also does not allow for the assignment of priorities between the concurrent processes involved. In the case study discussed, several considerations were made to achieve optimal performance:

- Since available memory is limited, all data was not loaded into memory at once; instead an iterative approach was taken by using a cursor.
- The task was split into subtasks by creating a main driver thread and two worker threads to handle the data and any errors in a decoupled manner.
- Data sharing was done via communication between all threads involved in a synchronized manner using channel based concurrency.
- Asynchronous progression of concurrent tasks was allowed with message buffering.
- Resource usage was optimized, but overuse was limited with the use of batching and limited buffer size.
- The state of all subtasks was carefully controlled by signaling (avoiding deadlock) and waiting on the graceful completion of all subtasks.

10. Conclusion

In closing notes, the article presented a demonstration of how concurrency mechanisms can be leveraged to achieve speed of execution and optimized resource usage within a software system. The fact that the system is handling large amounts of data or performing resource intensive tasks becomes irrelevant if it is designed with careful consideration and intelligent application of concurrency paradigms. The theory of communicating sequential processes, discussed at the beginning of the article, has been the primary influence behind channel based concurrency in programming languages like Go. Separation of concerns via concurrency paired with synchronized message passing and buffering allows concurrent tasks to progress simultaneously, thus benefiting overall execution speed. An in-depth discussion of

all the various concurrency paradigms is outside the scope of this discussion. The practical scenario discussed in this article provides insight into one of them - interprocess communication with message passing via channels. The application of the producer-consumer pattern is shown in the interaction of a sender and receiver process via channels. A message buffering process was demonstrated, facilitating the asynchronous interaction between sender and receiver processes. These concepts, when applied correctly, can vastly enhance the performance of production applications. This pattern has especially found usage in modern cloud based applications dealing with large amounts of data with a low desired latency.

References

[1] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM,* vol. 21, no. 8, pp. 666-677, 1985. [CrossRef] [Google Scholar] [Publisher Link]

[2] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe, "A Theory of Communicating Sequential Processes," *Journal of the ACM*, vol. 31, no. 3, pp. 560-599, 1984. [CrossRef] [Google Scholar] [Publisher Link]

[3] A.W. Roscoe, *The Theory and Practice of Concurrency,* Prentice-Hall, 1997. [Google Scholar] [Publisher Link]

[4] Range and Close, A Tour of Go. [Online]. Available: https://go.dev/tour/concurrency/4

[5] The Go Memory Model, 2022. [Online]. Available: https://go.dev/ref/mem

[6] Channel (Programming). [Online]. Available: https://en.wikipedia.org/wiki/Channel_(programming)

[7] Communicating Sequential Processes. [Online]. Available: https://en.wikipedia.org/wiki/Communicating_sequential_processes

[8] Actor Model. [Online]. Available: https://en.wikipedia.org/wiki/Actor_model

[9] Sql Package - database/sql - Go Packages, Go Standard Library Documentation, Version go1.22.3., 2024. [Online]. Available: https://pkg.go.dev/database/sql