PARALLELISM & CONCURRENCY STRATEGIES WITHIN PROGRAMMING LANGUAGES

Timothy DeVries

Calvin College
Department of Electrical and Computer Engineering
3201 Burton Street SE
Grand Rapids, MI 49546

November 30, 2015

Contents

1 Abstract

Nearly all computers made today have at least two cores. This means that nearly every standalone system today has the ability to execute code simultaneously. Even more so, with the advent of the internet and fast communication topologies, distributed systems of computing are becoming more common. Parallelism is not natural to many languages, at least in their original form, because it is not always deterministic, sequential or purely algorithmic. Languages such as Golang, Chapel and Erlang have attempted to solve that problem. There are also frameworks such as MPI and OpenMP that have been created that add on to original programming languages, such as C, and allow for concurrency and parallelism to be implemented.

These different structures each have strengths, weaknesses and ideal implementations. As with sequential programming, patterns are being developed for parallel programming that will allow developers to create code more easily. Ultimately, these strategies deserve study and reflection as parallelism appears to be the future of computing.

2 Introduction

The vast majority of processors shipped by Intel today are multi-core. This trend has been happening for over a decade. In 2007, "more than 70 percent of all server, desktop and mobile processors that Intel ships [were] multicore" [1]. This means that nearly all of modern computing entities have the ability to perform some form of parallelism or concurrency [1].

With so many devices now able to perform multiple operations at once, parallelization and its structure is becoming important in many different area of development, from web service handling to operating system multi-tasking. In this report, several broad strategies will be discussed, with implementations shown both for a sequential version of the program and a parallel or concurrent version.

3 Strategies within Programming Languages

3.1 General Strategy Outline

Each subsection will contain the following sections, which are shown with example topics:

General Description of Parallelism Strategy The background of each language will be discussed, including reasons for being created and problems hoped to address. Where applicable, the effectiveness of the language when attempting to solve the problems will be discussed.

Basic Example Using the Parallelism Strategy For each parallelism strategy, or sub-strategy, example code will be provided. Each set of code will address the same algorithm, regardless of the implementation. The algorithm that will be implemented in each language is summing the numbers from 0 to N and printing the result. For each implementation, a sequential example of the algorithm will be shown. This will be done so that one can see the transition from a simple sequential algorithm to a simple parallel or concurrent algorithm. After the sequential example has been shown, a basic parallel or concurrent algorithm will be constructed. The strengths and weaknesses of this implementation will be discussed, along with scalability of the language, either through size of the problem, number of processes.

Unique Characteristics & Hardware Implementations The discussion of the scalability of the solution will lead naturally into its unique characteristics and optimal hardware implementation. Items to be discussed will include multi-core vs. many-core (i.e. an 8-core notebook process vs. a GPU with 1024 cores), memory access by thread of execution and memory location.

3.2 Multi-process

Multi-processed implementations of concurrency are based on creating separate threads of execution. In this implementation, these threads of execution are called processes. Processes have no shared memory between other processes.

[1] Please refer to the appendix for a discussion concerning concurrency vs. parallelism.

The absence of shared memory is an advantage in some cases. For example, when passing a reference during the creation of Process A, if that variable is changed in the host process, the child process will read the changed result, rather than the original result. This will be discussed further in the Multi-threaded section. However, the lack of shared memory can also be a disadvantage. Complex systems of sharing information between processes can result in a slow of the process, along with redundancy of data or instructions.

Multi-process implementations are often useful for distributed systems because they do not assume the use of shared memory. However, they are able to be used in shared memory by allocating additional memory for each new process.

3.2.1 C with MPI

A common example of the Multi-process implementation of parallelism is C with a Message Passing Interface Library.

The Message Passing Interface (MPI) library is an implementation for parallelism that was adopted by many academic and industry partners. It is mainly used for parallelism, not concurrency. Each object has its own stack, heap, etc. Information must be passed to these objects. They do not inherit from "parent threads" or any other such object, which distinguishes the library from POSIX pthreads. It has implementations in several different languages. [2]

MPI is used so often in the multi-processing context that is often included with the standard Linux installations. It contains its own compiler and execution helper. The sequential and parallel implementations are shown in the following two sections.

Sequential C Code The implementation of the C code is very straightforward. It mainly revolves around the lines of code seen in **Listing 1** and does not require any of the functionality that MPI provides.

```
unsigned long long sum_range(unsigned long long min, unsigned long long max) {
    unsigned long long i, sum;
    sum = 0;                        //  Initialize our sum to 0
    for(i = min; i < max; i++) {    //  Loop from our minimum value to the maximum
        sum += i;                   //  Increase our sum
    }
    return sum;                     //  Return the sum
}
```

```
    finalSum = sum_range(0, maximum); // Start from 0 and go to the maximum (from command line)
```

Listing 1: "A Sequential Sum in C"

The algorithm above is sequential, and therefore the first process must iterate from zero to the maximum given in command line.

Parallel C Code There are two main sections used to convert the previous sequential C program to a parallel implementation. The first is initializing MPI and its processes. This can be seen in the first half of **Listing 2**. The MPI processes are created at runtime by the call to `mpirun`. An example call would be `mpirun -np 2 ./parallel_sum 10`, where `-np 2` represents the number of processors being set equal to two and `./parallel_sum 10` represents which binary to execute with its own command line arguments.

Because MPI creates separate processes, each must find its information by "asking" for it. By making the three queries in **Listing 2**, it finds out any pertinent command line arguments, how many other processes are executing simultaneously, and which process it is. Once it knows this information, the process knows all it needs to know to calculate its share of the sum.

This algorithm uses a *Reduction* pattern to combine the results from multiple processes back to the host process. This is necessary because they processes do not share memory, and therefore cannot modify some shared global variable or address.

```
    // Start our MPI items
    MPI_Init(&argc, &argv);                 //  Initialize MPI from command line arguments
    MPI_Comm_size(MPI_COMM_WORLD, &numProc); //  Find the number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &id);      //  Find out which process "we" are
```

```
    //  We will assign each PE it's own section of that section
    long double width = maximum / numProc;

    long double start = id * width;
    long double end = (id + 1) * width;

    localSum = sum_range(id * width, (id + 1) * width);

    MPI_Reduce(&localSum, &finalSum, 1, MPI_UNSIGNED_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
```

Listing 2: "A Parallel Sum in C with MPI"

The results of the two programs are shown below in **Listing 3**

```
C with MPI Results:
NumProc: 1, Time: 5.374006, Sum: 499999999500000000
NumProc: 2, Time: 0.354150, Sum: 499999999500000000
```

Listing 3: "Results for C with MPI"

The MPI library provides a standard way of creating processes. This abstraction provides the user with the ability to create a system that will work on both distributed and shared systems. This is because the structure of MPI does not rely on shared memory. This allows MPI to scale across multiple processing units without any change in the code. It will even run on the same binary.

However, the drawback, as with other multi-process structures is the need to copy any information that is in the stack or heap over to each process. This can be mitigated by only reading the data on a per-process level after initialization of the MPI processes. It functions well on a wide range of processing units, which can be changed at run time.

3.3 Actor Model

The actor model is a subset of the multi-process strategy, at least by the definition stated above. It falls into this category because the processing units do not use shared memory. In the actor model, an *Actor* is the "computational agent". This could be nearly any form of computational object, including memory chips, subprograms, or even entire computers. Whenever an actor communicates with another actor, it is called an *event*. However, this communication only refers to the act of receiving an event. All events are consider receiving events, there are no sending events.

The actor model was created to address the problem that "today's algorithmic programming languages were designed to express deterministic sequential algorithms". Due to this consideration at creation, the Actor model has several unique emphasises and patterns in its implementation. It primarily emphasises the use of communication that occurs while the computation is ongoing. This communication is similar to a "mail service". In the actor model's mail service, in that the message (or the mail) is always sent, but there is a variable amount of time until that message is received (or the mailbox is opened). [3]

It has many of the same characteristics of the general multi-process strategy, except that it often takes care of "little details" that are forgotten in generic multi-processing. It will remove the need for locks and semaphores in many situations, and provides optimal "under-the-hood" solutions to these problems.

3.3.1 Erlang

Erlang is a functional language developed to be used in the actor model. In Erlang's implementation, creating concurrency is very simple and is comparatively safe to some of the other methods discussed in this report. To create a new agent, one can simply call `spawn(module, function, listOfArgs)`, where the module is the location of the function that is to be called with the list of arguments. When this is a called, a new agent of execution is spawned as a process. It is called a process because it shares no data with the other threads of execution. Due to the lack of shared memory, it must use some sort of message passing to communicate with other processes. Erlang has a built-in message passing structure.

The messaging structure for Erlang has built-in fault tolerance. If an agent receives a message, it appends that message to a queue. Once reached in the queue, if it cannot be handled, it is left in the queue and the next item in the queue is handled. This relieves the programmer of the necessity of making many different types of locks and makes it so that locks do not have to handle when a connection is lost, or some other real-world problem that occurs during communication (particularly between distributed systems). To send a message in Erlang and take advantage of the built-in structure, one uses the following pattern of `pid ! message`.

As mentioned above, once an actor has received a message, it places it in a queue and must be handled [4].

```
receive
    pattern1 ->
        actions1;
    pattern2 ->
```

```
            actions2;
        ...
        patternN ->
            actionsN
    end.
```

Listing 4: "Erlang Receive Pattern"

The pattern shown in **Listing 4** will be used in the concurrent Erlang Example to pass messages between the different actors.

Sequential Erlang Example As mentioned above, Erlang is a functional programming language. As such, it does not have for loops. To simulate for loops, one must write a recursive function that does the work of the for loop. In **Listing 5**, we can see that function is called `find_sum` and it takes two arguments, `N`, which is the count of the numbers we will be summing over, and `Min`, which is the number at which the program will start summing. When the minimum is reached, it sums all of the numbers through the recursive calls and gives the sum.

```
find_sum(N, Min) ->
    if
        N == Min ->
            if
                Min == 0 ->
                    Min;
                Min > 0 ->
                    Min - 1
            end;
        N /= Min ->
            N - 1 + find_sum(N-1, Min)
    end.

print_sum(N, Min) ->
    StartTime = erlang:system_time(10000000),
    Result = find_sum(N, Min),
    EndTime = erlang:system_time(10000000),
    io:fwrite("NumProc: ~p, Time: ~p, Sum: ~p~n", [1, (EndTime - StartTime) / 10000000, Result]
    ).
```

Listing 5: "Sequential Sum in Erlang"

This sequential code does have a particularly unique feature: it does not use common programming constructs, such as a loop, that were relied on for the C code. This is why there is a recursive call in the place of the for loop that was seen in the similar C code.

Concurrent Erlang Example The concurrent implementation for Erlang is significantly more complex. As discussed above, there is no way to create loops except through recursion, so several helper functions must be made to keep track of how far down the recursion tree the program is at any time. Also, because actors do not have shared memory, all information must be passed between the actors and then eventually the main actor to compile the results. In the implementation shown in **Listing 6**, a Master-Worker approach is used.

The Master-Worker approach is where one thread of execution, the master, sends "jobs" to many other threads of execution. These jobs could have the same instructions with different data, different instructions with the same data, or some combination of those two options.

In the current example, the master spawns a master thread object that can receive the results from the sub-sums of the workers. It then spawns the children. Normally at this point, a for loop would be in order to spawn as many threads as the expect number of processors. However, a helper function was created to spawn the threads recursively until it "ran out" of threads to spawn. After all the threads have been spawned, the master receiver waits until it has received the specified number of threads. Upon receiving the last thread, it prints out the final result.

```
% Same recursive sum function as sequential
find_sum(N, Min) ->
    if
        N == Min ->
            if
                Min == 0 ->
                    Min;
                Min > 0 ->
                    Min - 1
            end;
        N /= Min ->
            N - 1 + find_sum(N-1, Min)
    end.

% Master Receiver
thread_total(OldMsg, Count, NumProc, StartTime) ->
    % Alert the user to how many threads are remaining
    % io:fwrite("Waiting for ~p threads~n", [NumProc - Count]),

    % Receive messages
    receive
        % Take any message (we are only passing one type)
        Msg ->
            % If we have enough messages sent
            if
                Count + 1 == NumProc ->
                    EndTime = erlang:system_time(10000000),
                    % Write the final sum
                    io:fwrite("NumProc: ~p, Time: ~p, Sum: ~p~n", [NumProc, (EndTime -
    StartTime)/10000000, Msg + OldMsg]);
                true ->
                    % Else, recursively call ourselves one closer
                    %   to the last call
                    thread_total(OldMsg + Msg, Count + 1, NumProc, StartTime)
            end
    end.

% Recursive function to spawn NumProc threads
spawn_thread(N, 0, NumProc, Master) ->
    % io:fwrite("Done Spawning for ~p maximum, ~p Processes and Master ID ~p~n~n",
    %           [N, NumProc, Master]);
    NumProc, Master, N;
spawn_thread(N, ID, NumProc, Master) ->
    % Spawn a new thread, and have it run the thread sum
    spawn(concurrent_sum, thread_sum, [(ID)*N/NumProc, (ID-1)*N/NumProc + 1, Master]),
    % Recursively call while decrementing our ID
    spawn_thread(N, ID - 1, NumProc, Master).

% Find the sum for a single Erlang thread
thread_sum(N, Min, MasterID) ->
```

```
    Result = find_sum(N, Min),                  % Sequentially find that sub-sum
    % io:fwrite("Max ~p and Min ~p: Result ~p~n",
    %           [N, Min, Result]),              % Show that it was found
    MasterID ! Result.                          % SEnd a message to the Master

% Thread called to find the sum: Entry Point
find_total_sum(N, NumProc) ->
    StartTime = erlang:system_time(10000000),
    Master = spawn(concurrent_sum, thread_total,
                   [0, 0, NumProc, StartTime]), % Spawn the master receiver
    spawn_thread(N, NumProc , NumProc, Master).  % Spawn all the worker threads
```

Listing 6: "Concurrent Sum in Erlang"

While this implementation is perhaps more difficult to understand than the C implementation above, it does provide inherent safety knowing that the actor will always receive the message and never have a race condition while writing.

The total line count grew from around 16 to approximately 60 to move from the sequential to parallel implementations of this algorithm. It also required significantly more design work, considering how to implement multiple recursive function calls and to return values at the correct time. However, this method gives the user very fine control over what each process would do, and is very safe compared to many other languages. There was no need to create a lock on the master receiver, which would be necessary if the user was using the standard POSIX thread library. This is because, as mentioned previously, every message is sent to the actor, but there is variable time until that actor "opens up the mail."

This becomes much more powerful in situations where communication is not as guaranteed as on one computer between processes on the same operating system. Imagine a website handling thousands of connections, which sometimes may fail or drop. If one of those connections previously held the lock for a key resource, and failed while holding the lock, it would be very difficult to retrieve that lock naturally. In the Actor model, there is no lock, and the master would just continue on down the queue of requests.

3.4 Multi-threaded

Multi-threaded languages use a parent and child relationship for the threads they create. A child will inherit all the information from a parent, and then have its own stack. This leads to several problems with non-deterministic results if information is stored or modified in the parent. However, it also allows for much less wasted memory and time copying over information from the parent's stack to the child's stack. Many threading libraries are based on the POSIX thread (pthread) standard.

3.4.1 Golang

Golang was originally developed in 2007 at Google. One of the cornerstones of the language was to support multiprocessing inherently and idiomatically. It does this primarily through goroutines. Goroutines are very easy to construct in golang. These goroutines spawn new psuedo-threads. They receive values from upstream via inbound channels and then perform

some function on this data, then send values downstream via outbound channels. This is an example of pipelining in Go [5].

However, goroutines are not exactly threads. They are, in short, a "function executing concurrently with other goroutines in the same address space". This is why they are often called threads, because of the shared memory with the host. Goroutines have several advantages, including being lightweight and handling their own stack and heap allocation and freeing [5]. They are lighter than threads, as they are multiplexed onto OS threads as required [6]. Not only are goroutines very lightweight in terms of hardware usage, they are also very easy to use in development. To add the concurrency, prefix the instruction with `go`. In otherwords, if the command was `functionCall(arg1, arg2)` then the new command would be `go functionCall (arg1, arg2)`.

Even so, Golang offers a more powerful tool for adding concurrency, which is a channel. Unlike goroutines, channels are able to signal completion. Channels can even be used in essentially sequential code to act as a return for a function.

Sequential Golang Example In **Listing 7**, the sequential implementation uses a channel that was created to pass return values back to the main function. Once a thread is created, and runs the function to `sum_range` function, that function will then pass the return value to the channel, which in turn passes it to the sum variable.

```
    c_seq := make(chan int)
    go sum_range(0, max, c_seq)
    sum_sequential = <- c_seq
```

Listing 7: "Sequential Sum in Go"

The sequential algorithm follows very closely with the C implementation.

Concurrent Golang Example However, channels begin to truly shine when concurrency is initiated in the program. Because one can wait for a specific channel at any time, it is possible to send off several goroutines one after another, and then wait for the results as necessary. In a practical application, this could look like sorting a list that was read from memory in a separate thread while the main thread continues to ask for more input. Then, right before the list is needed, the program can wait as necessary, but did not waste the time earlier sorting the list, but allowed a worker thread to do so.

In the current example, two threads are spawned with two channels listening for their output. Once they are both complete, the sum is then printed. This can be seen in **Listing 8**.

```
    c1 := make(chan int)
    c2 := make(chan int)
    go sum_range(0, max/2, c1)
    go sum_range(max/2, max, c2)

    sum_concurrent = <- c1
    sum_concurrent += <- c2
```

Listing 8: "Concurrent Sum in Go"

Golang provides a truly elegant way of providing concurrency to the program. It is considered concurrency because the goroutines can be executing any type of function – they are not required to perform the same action or instruction. Also, any function can be made into a concurrent call with the addition of the keyword `go` and can be assured to be complete using channels. Channels can act as locks, return functions and more. Yet, even with these more advanced tools, the problems with multi-threading still exist. Using channels or or goroutines, one is able to cause significant problems with the shared memory of channels if all race conditions are not handled.

3.5 OpenMP

OpenMP is an open source pragma-based framework. This is the slowest of the popular parallelization or concurrent frameworks, but also one of the simplest to develop in [7]. Pragmas are definitions inserted into code that will alert the compiler to do something different if the right libraries are included. Recall the sequential sum method in C shown in **Listing 1**. To parallelize this using OpenMP, simply add `#pragma omp parallel for`, which lets the compiler know that this should be parallelized. An example of this is shown in **Listing 9**.

```
    #pragma omp parallel for
2   for(i = min; i < max; i++) { //  Loop from our minimum value to the maximum
        sum += i;                //  Increase our sum
4   }
```

Listing 9: "Example of Pragma Based For Loop"

This is why OpenMP is considered one of the easiest frameworks to begin implementing parallel execution in a program. It is very powerful when dealing with legacy code that does not take advantage of multi-core processors at all and can be added with very little extra work. OpenMP has many different pragmas that can be used to create different types of parallelization. It can use reduction patterns, similar to what was seen in the MPI example above. There are several tools that will allow developers to analyze how long a program stays in a certain location, and OpenMP is often used in conjunction with them to provide large speed ups with little development effort.

However, this ease of use comes at a cost. There is often a 15 to 20 percent loss of performance compared to structures actually created in something like MPI.

3.6 Partitioned Global Address Space (PGAS)

PGAS are used for single program, multiple data programming. These programs have shared variables for communication instead of using message passing [8]. This is why they are often

associated with multi-threaded applications, as threads of execution will have shared memory, or at least the illusion of shared memory.

3.6.1 Chapel

Chapel is a language that was designed to once again combat the idea that many modern languages were constructed to solve deterministic, sequential algorithms. It's primary goal was to fix the problem that "the population of users who can effectively program parallel machines comprises only a small fraction of those who can effectively program traditional sequential computers, and this gap seems only to be widening as time passes" [9]. The language was originally developed and is still maintained by Cray Inc., a modern supercomputer manufacturer [10]. However, the language is open source and portable to non-Cray machines.

Chapel has four main goals that drive its ideology. The first is "General Parallelism". This means that the creators of the language want to provide the user with the ability to execute any parallel algorithm on any type of parallel hardware. The rest are "Separation of Parallelism and Locality, Multiresolution Design and Productivity Features" [10].

Sequential Chapel Example Chapel has the feeling of many other popular languages today. As can be seen in the sequential implementation shown in **Listing 10**,

```
use Time;

// Read in the command line argument, if passed
config var maximum: int = 10;

// Initialize our sum variable
var sum:int = 0;

// Declare and start our timer
var t:Timer;
t.start();

// Loop through until we have processed (maximum) number
for i in {0..(maximum-1)} {
    // Increment our sum
    sum = sum + i;
}

t.stop();

// Write out the result
writef("NumProc: 1, Time: %7.7r, Sum: %i\n", t.elapsed(), sum);
```

Listing 10: "Example of Chapel Sequential Sum"

This once again follows very closely to the implementation for the sequential sum in C.

Parallel Chapel Example Once again, a reduction pattern will be used in this example. Chapel provides a very simple interface for several different kinds of common parallel patterns. To invoke a parallel pattern in Chapel, insert the following template:

`<operator> <pattern_name> ( <objects_to_iterate_over> )`
This can be seen in line 27 of **Listing 11**.

```
use Time;

// Read in the command line argument, if passed
config var maximum: int = 10;
config var numProc: int = 1;

// Get the lsit of processor IDs
var processor_list: domain(1) = 1..numProc;

proc sum_range(min, max) {
    var sum: int = 0;
    var i: int = min;
    while (i < max) {
        sum = sum + i;
        i += 1;
    }
    return sum;
}

// Create the timer
var t:Timer;

// Start the timer
t.start();

// Loop through until we have processed (maximum) number
var sum = + reduce ([i in processor_list] sum_range( (i - 1) * maximum / numProc, i * maximum /
     numProc ));

// Finish the timer
t.stop();

// Write out the result
writef("NumProc: %i, Time: %7.7r, Sum: %i\n", numProc, t.elapsed(), sum);
```

Listing 11: "Example of Chapel Parallel Sum"

Chapel's interface is very easy to use for many of the common parallel strategies. It does a good job of handling the shared memory between agents, allowing for efficient handling of large amounts of data simultaneously.

4 Other Interesting Strategies

This section includes information that was throughout the research phase of this paper, that while interesting, was not feasible to include in this document.

4.1 CUDA

A proprietary, but free, implementation of parallelism that uses General Purpose Graphics Processing Units (GPGPUs) to perform - in general - vector problems. This is an example of a language that is primarily strongest with a Single Instruction, Multiple Data (SIMD) approach.

It can only be used with Nvidia GPUs, not with standard CPU cores or other GPUs (for example AMD). This allows a program to take advantage of the large amount of processing units in a GPU, sometimes more than 1024 cores in one GPU. It often uses vector processing [11].

4.2 OpenCL

An open-source alternative to CUDA. It can use any core on a CPU, not just a "CUDA-Core". However, because it is so generic, it is often much harder to configure and optimize. Because of this, it often suffers a 15-20% reduction in performance when compared to CUDA.

5 Conclusion

This report has found that there are many different implementations for parallelism and there is no "one-size-fits-all" approach. Just as in sequential programming, the need to balance rapid development and prototyping with speed often occurs. For example, in sequential programming one might want to use Python to create a system, but because of the amount of calculations, C might be a more viable choice. Similarly, in parallel programming one might want to use OpenMP to parallelize the system because of the short development time, but might need the extra speed that MPI can offer.

Not only are time considerations important, but hardware implementations play a key role in choosing a strategy. If the system will be distributed, then perhaps Erlang with its natural safeguards against dropped messages, out-of-order receiving, and pre-made actors might be the best choice. Or if the system will used shared memory, and the main program has already read in a large amount of data, then perhaps a multi-threaded approach might be best so that the data would not need to be copied for each thread.

In practice, time for consideration must be made before deciding on a language, implementation and pattern before development begins. However, it is not only important to spend time considering the options, but they must be well known to the user. I felt that I learned a lot from doing this research paper and I hope that I am able to make more educated decisions about parallelism in the future because of it.

6 Appendix

6.1 Concurrency vs. Parallelism

Because there is a common misconception that concurrency and parallelism are synonymous, both terms will be defined here and used accordingly throughout the paper. Concurrency is "programming as the composition of independently executing processes," while parallelism is "the simultaneous execution of (possibly related) computations" [6].

To elaborate, concurrency is about *handling* lots of different things at one time, while parallelism is about *doing* lots of things at once. In general, this means that concurrency is about the structure of a solution, language or abstraction while parallelism is about the actual execution of the solution.

6.2 Code Implementations

All code in the following sections was written by the author of this report, T.J. DeVries.

The results of all the implementations are shown below. Erlang was having a problem with big numbers, so it just runs 1,000,000.

```
================================================================================

                        Final Results, N = 1000000000
================================================================================

C with MPI Results:
NumProc: 1, Time: 5.374006, Sum: 499999999500000000
NumProc: 2, Time: 0.354150, Sum: 499999999500000000
Golang Results:
NumProc: 1, Time: 0.68977, Sum: 499999999500000000
NumProc: 2, Time: 0.67285, Sum: 499999999500000000
Chapel Results:
NumProc: 1, Time: 7.463785, Sum: 499999999500000000
NumProc: 2, Time: 4.231605, Sum: 499999999500000000
OpenMP Results:
NumProc 1, Time:  4.895306, Sum: 499999999500000000
NumProc 1, Time:  13.0191, Sum: 302028848749388210
Erlang Results:
NumProc: 1, Time: 0.1753617, Sum: 499999500000
NumProc: 2, Time: 0.4037933, Sum: 4.999995e11
```

Listing 12: "Final Results"

6.2.1 C Implementations

```
/* sequential_sum.c sums the numbers from 1 to N.
 * TJ DeVries, for ENGR 325 at Calvin College.
 */
```

```c

#include <stdio.h>      // printf(), etc.
#include <stdlib.h>     // exit()
#include <math.h>       // sqrt()
#include <time.h>   // Timer

/* retrieve desired maximum from commandline arguments
 * parameters: argc: the argument count
 *             argv: the argument vector
 * return: the number of trapezoids to be used.
 */
unsigned long long processCommandLine(int argc, char** argv) {
   if (argc == 1) { return 10; }
   else if (argc == 2) { return strtoull( argv[1], 0, 10 ); }
   else {
       fprintf(stderr, "Usage: ./parallel_sum [maximum]");
       exit(1);
   }
}

unsigned long long sum_range(unsigned long long min, unsigned long long max) {
    unsigned long long i, sum;
    sum = 0;                        //  Initialize our sum to 0
    for(i = min; i < max; i++) { //  Loop from our minimum value to the maximum
        sum += i;                   //  Increase our sum
    }
    return sum;                     //  Return the sum
}

int main(int argc, char** argv) {
    // Final sum
    unsigned long long finalSum;

    // Timing values
    clock_t startTime, stopTime;

    // Start the timer
    startTime = clock();

    // Get the maximum N
    unsigned long long maximum = processCommandLine(argc, argv);

    finalSum = sum_range(0, maximum); // Start from 0 and go to the maximum (from command line)

    stopTime = clock();

    printf("NumProc: %d, Time: %7f, Sum: %llu\n", \
            1, (double)(stopTime - startTime) / CLOCKS_PER_SEC, finalSum);

    return 0;
}
```

Listing 13: "A Sequential Sum in C with MPI"

```c
/* parallel_sum.c sums the numbers from 1 to N.
 * TJ DeVries, for ENGR 325 at Calvin College.
 */

#include <stdio.h>      // printf(), etc.
#include <stdlib.h>     // exit()
#include <math.h>       // sqrt()
#include <mpi/mpi.h>
```

```
9
/* retrieve desired maximum from commandline arguments
11  * parameters: argc: the argument count
 *             argv: the argument vector
13  * return: the number of trapezoids to be used.
 */
15 unsigned long long processCommandLine(int argc, char** argv) {
   if (argc == 1) { return 10; }
17   else if (argc == 2) { return strtoull( argv[1], 0, 10 ); }
   else {
19       fprintf(stderr, "Usage: ./parallel_sum [maximum]");
       exit(1);
21   }
}
23
unsigned long long sum_range(unsigned long long min, unsigned long long max) {
25     unsigned long long i, sum;
    sum = 0;
27
    for(i = min; i < max; i++) {
29       sum += i;
    }
31
    return sum;
33 }

35 int main(int argc, char** argv) {
    unsigned long long localSum = 0;
37     unsigned long long finalSum;

39     // Timing values
    double startTime, stopTime;
41
    int numProc = -1;
43     int id = -1;

45     // Start our MPI items
    MPI_Init(&argc, &argv);                  //  Initialize MPI from command line arguments
47     MPI_Comm_size(MPI_COMM_WORLD, &numProc); //  Find the number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &id);      //  Find out which process "we" are
49
    // Start the timer
51     startTime = MPI_Wtime();

53     // Get the maximum N
    unsigned long long maximum = processCommandLine(argc, argv);
55
    //  We will assign each PE it's own section of that section
57     long double width = maximum / numProc;

59     long double start = id * width;
    long double end = (id + 1) * width;
61
    localSum = sum_range(id * width, (id + 1) * width);
63
    MPI_Reduce(&localSum, &finalSum, 1, MPI_UNSIGNED_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
65
    stopTime = MPI_Wtime();
67
    if (id == 0) {
69       printf("NumProc: %d, Time: %7f, Sum: %llu\n",
              numProc, stopTime - startTime, finalSum);
71     }

73     MPI_Finalize();
```

```
    return 0;
}
```

Listing 14: "A Parallel Sum with MPI"

```
/* sequential_sum.c sums the numbers from 1 to N.
 * TJ DeVries, for ENGR 325 at Calvin College.
 */

#include <stdio.h>      // printf(), etc.
#include <stdlib.h>     // exit()
#include <math.h>       // sqrt()
#include <time.h>   // Timer
#include <omp.h>

/* retrieve desired maximum from commandline arguments
 * parameters: argc: the argument count
 *             argv: the argument vector
 * return: the number of trapezoids to be used.
 */
unsigned long long processCommandLine(int argc, char** argv) {
   if (argc == 1) { return 10; }
   else if (argc == 2) { return strtoull( argv[1], 0, 10 ); }
   else {
       fprintf(stderr, "Usage: ./parallel_sum [maximum]");
       exit(1);
   }
}

unsigned long long sum_range(unsigned long long min, unsigned long long max) {
    unsigned long long i, sum;
    sum = 0;                    //  Initialize our sum to 0
    #pragma omp parallel for
    for(i = min; i < max; i++) { //  Loop from our minimum value to the maximum
        sum += i;               //  Increase our sum
    }
    return sum;                 //  Return the sum
}

int main(int argc, char** argv) {
    // Final sum
    unsigned long long finalSum;

    // Timing values
    clock_t startTime, stopTime;

    // Start the timer
    startTime = clock();

    // Get the maximum N
    unsigned long long maximum = processCommandLine(argc, argv);

    finalSum = sum_range(0, maximum); // Start from 0 and go to the maximum (from command line)

    stopTime = clock();

    printf("NumProc %i, Time:  %7.7g, Sum: %llu\n", \
            omp_get_num_threads(), (double)(stopTime - startTime) / CLOCKS_PER_SEC, finalSum);

    return 0;
}
```

Listing 15: “A Parallel Sum with Omp”

6.2.2 Erlang Implementations

```
-module(sequential_sum).
-export([find_sum/2, print_sum/2]).

find_sum(N, Min) ->
    if
        N == Min ->
            if
                Min == 0 ->
                    Min;
                Min > 0 ->
                    Min - 1
            end;
        N /= Min ->
            N - 1 + find_sum(N-1, Min)
    end.

print_sum(N, Min) ->
    StartTime = erlang:system_time(10000000),
    Result = find_sum(N, Min),
    EndTime = erlang:system_time(10000000),
    io:fwrite("NumProc: ~p, Time: ~p, Sum: ~p~n", [1, (EndTime - StartTime) / 10000000, Result]
    ).
```

Listing 16: “Sequential Sum in Erlang”

```
-module(concurrent_sum).
-export([find_sum/2, find_total_sum/2, thread_sum/3, thread_total/4]).

% Same recursive sum function as sequential
find_sum(N, Min) ->
    if
        N == Min ->
            if
                Min == 0 ->
                    Min;
                Min > 0 ->
                    Min - 1
            end;
        N /= Min ->
            N - 1 + find_sum(N-1, Min)
    end.

% Master Receiver
thread_total(OldMsg, Count, NumProc, StartTime) ->
    % Alert the user to how many threads are remaining
    % io:fwrite("Waiting for ~p threads~n", [NumProc - Count]),

    % Receive messages
    receive
        % Take any message (we are only passing one type)
```

```
        Msg ->
            % If we have enough messages sent
            if
                Count + 1 == NumProc ->
                    EndTime = erlang:system_time(10000000),
                    % Write the final sum
                    io:fwrite("NumProc: ~p, Time: ~p, Sum: ~p~n", [NumProc, (EndTime -
    StartTime)/10000000, Msg + OldMsg]);
                true ->
                    % Else, recursively call ourselves one closer
                    %   to the last call
                    thread_total(OldMsg + Msg, Count + 1, NumProc, StartTime)
            end
    end.

% Recursive function to spawn NumProc threads
spawn_thread(N, 0, NumProc, Master) ->
    % io:fwrite("Done Spawning for ~p maximum, ~p Processes and Master ID ~p~n~n",
    %           [N, NumProc, Master]);
    NumProc, Master, N;
spawn_thread(N, ID, NumProc, Master) ->
    % Spawn a new thread, and have it run the thread sum
    spawn(concurrent_sum, thread_sum, [(ID)*N/NumProc, (ID-1)*N/NumProc + 1, Master]),
    % Recursively call while decrementing our ID
    spawn_thread(N, ID - 1, NumProc, Master).

% Find the sum for a single Erlang thread
thread_sum(N, Min, MasterID) ->
    Result = find_sum(N, Min),                  % Sequentially find that sub-sum
    % io:fwrite("Max ~p and Min ~p: Result ~p~n",
    %           [N, Min, Result]),                % Show that it was found
    MasterID ! Result.                          % SEnd a message to the Master

% Thread called to find the sum: Entry Point
find_total_sum(N, NumProc) ->
    StartTime = erlang:system_time(10000000),
    Master = spawn(concurrent_sum, thread_total,
                   [0, 0, NumProc, StartTime]), % Spawn the master receiver
    spawn_thread(N, NumProc , NumProc, Master).  % Spawn all the worker threads
```

Listing 17: "Concurrent Sum in Erlang"

6.2.3 Golang Implementations

```
// For loop

package main

import (
    "fmt"
    "flag"
    "time"
)

func sum_range(min, max int, c chan int)  {
    var i int
    var temp_sum int

    temp_sum = 0
```

```
    for i = min; i < max; i++ {
        temp_sum += i
    }

    c <- temp_sum
}

func main() {
    var sum_sequential, sum_concurrent int
    var max int

    flag.IntVar(&max, "max", 10, "Number of integers to sum")
    flag.Parse()

    start_sequential := time.Now()

    c_seq := make(chan int)
    go sum_range(0, max, c_seq)
    sum_sequential = <- c_seq

    elapsed_sequential := time.Since(start_sequential)
    fmt.Printf("NumProc: %d, Time: %7.5f, Sum: %d\n", 1, float64(elapsed_sequential)
    /1000000000, sum_sequential)

    start_concurrent := time.Now()

    sum_concurrent = 0

    c1 := make(chan int)
    c2 := make(chan int)
    go sum_range(0, max/2, c1)
    go sum_range(max/2, max, c2)

    sum_concurrent = <- c1
    sum_concurrent += <- c2

    elapsed_concurrent := time.Since(start_concurrent)
    fmt.Printf("NumProc: %d, Time: %7.5f, Sum: %d\n", 2, float64(elapsed_concurrent)
    /1000000000, sum_concurrent)
}
```

Listing 18: "Parallel and Concurrent Sum in Go"

6.2.4 Chapel Implementations

```
use Time;

// Read in the command line argument, if passed
config var maximum: int = 10;

// Initialize our sum variable
var sum:int = 0;

// Declare and start our timer
var t:Timer;
t.start();

// Loop through until we have processed (maximum) number
for i in {0..(maximum-1)} {
    // Increment our sum
```

```
    sum = sum + i;
}

t.stop();

// Write out the result
writef("NumProc: 1, Time: %7.7r, Sum: %i\n", t.elapsed(), sum);
```

Listing 19: "Example of Chapel Sequential Sum"

```
use Time;

// Read in the command line argument, if passed
config var maximum: int = 10;

// Initialize our sum variable
var sum:int = 0;

// Declare and start our timer
var t:Timer;
t.start();

// Loop through until we have processed (maximum) number
for i in {0..(maximum-1)} {
    // Increment our sum
    sum = sum + i;
}

t.stop();

// Write out the result
writef("NumProc: 1, Time: %7.7r, Sum: %i\n", t.elapsed(), sum);
```

Listing 20: "Example of Chapel ParallelSum"

References

[1] W. S. Gary Carleton, "Performance analysis and multicore processors." http://www.drdobbs.com/tools/performance-analysis-and-multicore-proce/184417069, 2006.

[2] J. Poole, "Implementation of a hardware-optimized mpi library for the scmp multiprocessor." http://scholar.lib.vt.edu/theses/available/etd-08112004-210027/unrestricted/PooleThesis.pdf, 2004.

[3] W. D. Clinger, "Foundations of actor semantics." https://dspace.mit.edu/bitstream/handle/1721.1/6935/AITR-633.pdf?sequence=2.

[4] Erlang, "Getting started with erlang." http://www.erlang.org/doc/.

[5] G. Documentation, "Effective go." https://golang.org/doc/effective_go.html.

[6] R. Pike, "Concurrency is not parallelism." http://talks.golang.org/2012/waza.slide#1, January 11, 2012.

[7] "Openmp." https://www-new.comp.nus.edu.sg/~wongwf/papers/fpt06.pdf.

[8] "Cray." http://www.gwu.edu/~upc/publications/ppopp05.pdf.

[9] H. P. Z. B. L. Chamberlain, D. Callahan, "Parallel programmability and the chapel language," September 2007.

[10] B. Chamberlain, "Chapel: Productive parallel computing." http://www.cray.com/blog/chapel-productive-parallel-programming/, May 2013.

[11] "Cuda." https://developer.nvidia.com/about-cuda.