Report

Boxuan Yang

April 20, 2024

1 Deadlock

1.1 Threshold of N to cause deadlock

The deadlock threshold for N is found using with the below binary search. The minimum N that can cause deadlock is 32768.

The pseudocode is attached below.

```
upper = 1000000
lower = 1000

while (upper - lower) > 1  do:
    mean = (upper + lower)/2

    if(deadlock occurs at mean):
        upper = mean
    else
        lower = mean
```

Using the above binary search, the minimum N that can cause deadlock is 32768.

1.2 Reasons of Deadlock

From MPI lecture, we know that when the message size is large enough, the MPI_Send may switch to synchronous mode, which means MPI_Send does not return until the destination process has received the message and hence deadlock occurs when N gets too large.

I will illustrate it with the example below.

Suppose we have 2 process, P0 and P1. From the provided code, what P0 will do is: 1. Send to P1 2. Receive from P1 3. Send to P1 4. Receive from P1. What P1 will do is: 1. Send to P0 2. Receive from P0 3. Send to P0 4. Receive from P0. When the message is small, P0 will go to step 2 to receive message from P1 after the message is copied to the internal buffer. However, when the message is large such that the internal buffer is fully occupied, it will not move to step 2 until P1 has successfully received the message. But we can see that the first step of P1 is to send to P0 and P1 is also waiting for P0 to successfully receive message. Hence, a deadlock occurs.

1.3 Fix the Deadlock

We notice that in the provided code, the deadlock occurs due to the asynchrounsness of the send's and receive's. Hence, we can fix the deadlock by re-ordering the send and receives.

The pseudocode is provided below.

```
if(rank % 2 == 0){
    Send to the top process
    Receive from the bottom process
    Send to the bottom process
    Receive from the top process
}else{
    Receive from the bottom process
    Send to the top process
    Receive from the top process
    Send to the bottom process
}
```

Detailed implementation can be found in my update_boundary(double u, int ldu) function.

2 Non-blocking analysis

To evaluate the performance, I performed tests with M=10000, N=10000, r=100 with and p values of 2, 4, 6, 8 and 10. That is, mpirun -np p ./testAdvect 10000 10000 where $p \in \{2, 4, 6, 8, 10\}$

p	2	4	6	8	10
Non-blocking	15.9s	10.1s	9.01s	8.64s	8.52s
Reordering	15.9s	10.2s	9.09s	8.64s	8.53s

From the above table we can see that Non-blocking method outperforms the Reordering method by a tiny margin. Hence, we will use non-blocking method for the rest of this assignment.

The detailed implementation of non-blocking communication can be found in my updateBoundry function.

3 Performance modelling and calibration

3.1 Performance model

Since per element computation time is not defined in lecture, we will assume in this case t_f means the time it takes to perform one floating number arithmetic operation. We will also assume a double data corresponds to a word.

We can write total time $t = t_{comm} + t_{compute}$.

Given that we need to perform r number of iterations for each process and we need to have 4 communication for each iteration. The size of each message is N. Therefore, we can write $t_{comm} = r \times P(4t_s + 4Nt_w)$.

```
for(int i = 0; i < m; i++)
  for(int j = 0; j < n; j++){
    v[i*ldv+j] =
    cim1 *(cjm1*u[(i-1)*ldu+j-1]+cj0*u[(i-1)*ldu+j]+cjp1*u[(i-1)*
    ldu + j+1])
+   ci0 *(cjm1*u[i*ldu+j-1] + cj0*u[i*ldu+j] + cjp1*u[i*ldu+j+1])
+   cip1 *(cjm1*u[(i+1)*ldu+j-1] + cj0*u[(i+1)*ldu+j]+cjp1*u[(i+1)*
    ldu+j+1]);
  }
```

Above is the for loop part in the update_advection_field function. After carefully investigation, we found each iteration in the inner loop requires 20 floating operations(12 for multiplication and 8 for addition). We notice there are other floating point operations in the update_advection_field function too. But they all occur before the loop and therefore can be ignored. Hence, we can model $t_{compute}$ by

$$t_{compute} = r \times (20 M_{loc} N_{loc})$$

,where

$$M_{loc} = \frac{M}{P}, N_{loc} = N$$

.

Therefore,

$$t = t_{comm} + t_{compute}$$

,where

$$t_{comm} = r \times P \times 4 \times (t_s + N t_w)$$

$$t_{compute} = r \times 20(\frac{MN}{P}) \times t_f$$

3.2 Experiments

3.2.1 tf

To measure t_f, we measure the time of performing 20000000 times of double arithmetic operations and divide it by 20000000 to amortize the measurement error. The detailed implementation can be found in tf.c program. We found tf to be 2.67×10^{-9}

3.2.2 ts

To measure t_s, we measure the time of sending 10000 messages with null as buffer pointer. In this way, the communication time is made up of startup time only and we divide the time by 10000 to amortize the measurement error. The detailed implementation can be found in tw.c program. We found t_s to be 6.23×10^{-8}

3.2.3 tw

To measure t_w, we measure the time of sending 10000 messages with 10000 number of words for each message. In this way, the total time $t = 10000 \times (t_s + 10000 \times t_w)$. Then we use the t_s from above to calculate the t_w. The detailed implementation can be found in tw.c program. We found $t_w = 7.64 \times 10^{-10}$

t_w	7.64×10^{-10}s
t_f	2.67×10^{-9}s
t_s	6.23×10^{-8}s

3.3 Strong scaling analysis

3.3.1 Theoretical analysis and time prediction

From the formula in section 3.1, we know:

$$t = r \times (4t_s + 4Nt_w + 14N\frac{M}{P} \times t_f)$$

.

We can see that from as P goes to infinity, the execution time approaches $r \times (4t_s + 4Nt_w)$, which is 3.3×10^{-4}

Substitute the t_s, t_w and t_f, we can calculate the expected execution time presented in the table in 3.3.2.

3.3.2 Emperical analysis

From the perfModels lecture, strong scaling assumes fixed problem size. Therefore, we need to evaluate the performance with a fixed problem size and varying p's.

We evaluate the performance with mpirun -np p ./testAdvect 1000 1000 100, with $p \in \{1, 4, 8, 12, 90\}$

At first I use M, N = 5000, but later on I reduce M, N to 1000 to save computation resource. The result of strong scaling analysis is presented below.

p	Actual Time	Predicted time
1	0.23s	3.74s
4	0.063s	0.94s
8	0.036s	0.47s
16	0.017s	0.23s
32	0.025s	0.12s
64	0.02s	0.059s
128	0.03s	0.03s

We can see from the above table that the discrepancy between prediction and actual time is fairly large for $p < 128$. I suspect this is due to 1. the measurement error in the MPI_Wtime function 2. the above calculation did not take the instruction-level parallelism of pipelining inside the CPU into consideration, which makes program runs much faster than theoretical prediction.

4 The effect of 2D process grids

4.1 Code extension

My code for 2D extension can be found in init_parallel_parameter_values() and update_boundary() functions.

4.2 New performance model

Under 2D process grid, each process needs to have 8 communications with neighbouring process. 4 of which are with top process and bottom process with N_{loc} as message size. The other 4 are with left and right process with with $M_{loc} + 2$ as message size. Therefore, the communication time can be modeled as

$$t_{comm} = r \times (8t_s + 4(M_{loc} + 2 + N_{loc}) \times t_w)$$

, where

$$M_{loc} = \frac{M}{P}, N_{loc} = \frac{N}{Q}, P \times Q = p$$

For the computation time, each process needs to perform $r \times M_{loc}N_{loc}$ of floating point operations, hence, computation time can be modeled as:

$$t_{computation} = r \times (M_{loc} \times N_{loc})$$

Therefore, total time is:

$$t = 8r \times t_s + 4r(\frac{M}{P} + 2 + \frac{N}{Q})t_w + r \times \frac{MN}{p}$$

, where p is the total number of processes. Given that $p = P \times Q$, the above equation can be further reduced to:

$$t = 8r \times t_s + 4r(\frac{MQ}{p} + 2 + \frac{N}{Q})t_w + r \times \frac{MN}{p}$$

4.3 Experiment on one node

We conduct experiment with M=N=1000, r=100 and np=24 on a single node with P,Q pair to be (1, 24), (2, 12), (3, 8), (4, 6), (6, 4), (8, 3), (12, 2), (24, 1). The result is presented in the table below.

(P, Q)	Execution time
(1, 24)	1.97e-02s
(2, 12)	2.44e-02s
(3, 8)	1.89e-02s
(4, 6)	2.34e-02s
(6, 4)	1.37e-02s
(8, 3)	2.20e-02s
(12, 2)	1.83e-02s
(24, 1)	2.10e-02s

From the above table, we can see square ratio indeed has a different effect to the default. We observe that performance can be greater than or lower than Q=1.

4.3.1 Optimal ratio

We can take the derivative of $\frac{dt}{dQ}$ of the performance model in 4.2, set it to 0 we get $Q = \sqrt{p}$ when M=N. Therefore, optimal Q should be $\sqrt{24} \sim 5$. However, Q=5 is not possible here because 24 is not divisible by 5. Therefore, we observe the optimal performance occurs at Q=6 where it is right next to 5. Regarding to t_w becomes 10x larger, we can see from the equation in 4.2 that this will not affect the optimal ratio.

4.4 Experiment on 2 nodes

We conducted experiment on 2 nodes with M=N=1000, r=100 and np=96. The results are presented below.

(P, Q)	Execution time
(1, 96)	2.56e-02s
(2, 48)	3.46e-02s
(3, 32)	1.74e-02s
(4, 24)	3.92e-02s
(6, 16)	4.28e-02s
(8, 12)	1.86e-02s
(16, 6)	1.89e-02s

5 Overlapping communication with computation

5.1 How overlapping works

I only implement overlapping for Q=1.

The overlapping technique works by exploiting the non-blocking mechanism of MPI_Isend and MPI_Irecv. In contrast to MPI_Send which does not return until certain operations are completed, MPI_Isend returns immediately after the call is initiated and leave the communication happening in the backend while continue on the following computation tasks. Therefore, we can implement overlapping by first performing the boundary updating job with non-blocking communication(MPI_Isend and MPI_Irecv). Then, we run advection for the inner halo such that inner halo advection can be performed in parallel with communication. In the next step, we add an MPI_Waitall function call to ensure that all communications have been completed. Finally, we update the upper and lower boundary row manually. Detailed implementation can be found in my run_parallel_advection_with_comp_comm_overlap function.

5.2 Performance impact & experiment

Overlapping will increase the performance because we can run the computation and communication at the same time.

Given that the communication is run in parallel with computation and our Q=1 , hence, our model in 4.2 can be rewritten as:

$$t = r \times (Max(4t_s + 4Nt_w, 20(M_{loc} - 2)Nt_f) + 20 \times 2 \times N)$$

,where $M_{loc} = \frac{M}{p}$

Therefore,

$$t = r \times (Max(4t_s + 4Nt_w, 20(\frac{M}{p} - 2)Nt_f) + 20 \times 2 \times N)$$

Experiment is conducted with M=N=1000, r=100 on 2 nodes. The result is presented in the table below.

p	non-overlap	overlap
2	1.29e-01s	1.24e-01s
4	8.17e-02s	6.58e-02s
8	3.57e-02s	3.21e-02s
16	1.68e-02s	1.43e-02s
32	9.32e-03s	1.43e-02s
64	1.58e-02s	1.81e-02s

We can see that overlapping outperforms non-overlap in most scenarios. We also notice overlapping can be less effective when $p \geq 32$. I suspect this occurs because our M, N are not big enough, therefore when p becomes large enough, the overhead computation in update_advection function(i.e., coefficients calculation) cannot be amortized by the relatively small values of M_{loc}, N_{loc}.

5.3 Difficulties for 2D process girds

Overlapping for 2D process grid is difficult because the left/right halo exchange is dependent on the top/bottom halo exchange, which means left/right communication cannot be overlapped and that can increase complexity for the problem.

6 Wide halo transfer

6.1 How wide halo transfer works

By introducing a wide of width greater than one, we can update the local advection iteratively by first update (m+2*w-2)x(n+2*w-2), (m+2*w-4)x(n+2*w-4), ..., (m)x(n). We can notice that with such technique, we only need to perform communication per w times of local computaion. Hence, the communication cost can be greatly reduced.

6.2 Advantages and Performance impact

The main advantage of wide halo is that the number of communications has been reduced from r to $\frac{r}{w}$. Our performance model can be rewritten as:

$$t = \frac{8rt_s w}{w} + 4r(\frac{NP}{p} + \frac{M}{P})t_w + \sum_{i=0}^{w-1} \frac{20r}{w}(\frac{M}{P} + 2i)(\frac{NP}{p} + 2i)t_f$$

6.3 Experiment

I run experiment with M=N=1000, r=100, P=10, with w=1 and 2 on 2 nodes. Result is shown below.

w	1	2
	2.91e-02s	2.15e-02s height

We can see from the table above that the performance indeed increase as w goes from 1 to 2.

7 Literature Review

7.1 Diamond tiling

7.1.1 Motivation

Tiling is a common technique to optimize stencil computations by exploiting the data locality and parallelism from stencil computations. Tiling is defined by the tile shape, which is derived from the directions selected to slice iteration spaces of statements represented by tiling hyperplanes, as well as the tile size. How to discover the optimal tile size and shape is essential to performance and is still the frontier of parallel computing research. However, the existing methods often lead to pipelined startup and load imbalance, which means not all processors are busy during parallelized execution and the work distribution is uneven across different processes. The author proposed a new tiling technique called diamond tiling to address this issue. Diamond tiling method ensures concurrent startup and perfect load balance whenever possible, which greatly improves performance. [1]

7.1.2 Effectiveness

According to the author, diamond tiling is able to outcompete a tuned domain-specific stencil code generator by 10-40%, and achived a speedup to previous compiler techniques of 1.3x to 10.1x.

7.2 Overlapped tiling

7.2.1 Motivation

In stencil computations, tiling is a very common technique used to enhance performance by exploiting data locality and parallelism. However, such method inevitably involves shadow regions, which can be a challenging programming task for the programmers. Therefore, to resolve this issue, the author proposed a method called overlapped tiling which is highly effective in supporting shadow region in a flexible, convenient and effective manner with the support of hierarchically tiled array(HTA) data structure. The purpose of HTA is to provide the programmer with natural representation for sequential and parallel tiled computations. Overlapped tiling works by providing abstractions of the shadow regions in the program and hence relieve the programmer from the burden of managing the detailed of the definition and update of these regions. [2][3]

7.2.2 Effectiveness

According to the author, the method is able to reduce the number of communication statements by 78% on average.

8 Performance outcome via combination of optimization techniques

The goal of the diamond tiling is to generate concurrent startup and balanced work distribution across processes, which can improve cache locality because too much workload may involve accessing remote data and hence decreases cache hit rate.

Overlapped Tiling can also enhance data locality. This technique is implemented using the hierarchically tiled array(HTA) data structure, which is a data structure specifically designed to deliver high data locality. It achieves this by organizing data into tiles such that data accessed together by the program are stored close to each other in memory, which improves spatial locality and increases cache hit rate. In terms of temporal locality, HTAs partition computations into iterations or time steps. During each iteration, computations update data within tiles. Through reusing data from previous iterations in the same tiles, HTAs exploit temporal locality.

In conclusion, the combination of the above techniques will enhance data locality. However, there are also some trade-offs. The first is implementation, combining the 2 methods poses extra complexity in the program, especially regarding to data dependency management. Second is overhead issue, combination of diamond tiling and overlapped tiling may introduce additional performance overhead due to increased challenges in managing overlapped regions.

9 Optimization implementation

I implement the optimization by exploiting thread level parallelism in copy_field function. I used OpenMP method to perform data copying, hence reduce the performance of copy_field function by Yx, where Y represents the number of available threads. The detailed implementation can be found in parAdvect.c file.

References: [1]: U. Bondhugula, V. Bandishti and I. Pananilath, "Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 5, pp. 1285-1298, 1 May 2017, doi: 10.1109/TPDS.2016.2615094. keywords: Face;Diamond;Parallel processing;Shape; Indexes;Silicon;Optimization;Compilers;program transformation;loop tiling; parallelism;locality;stencils

[2]: Guo J, Bikshandi G, Fraguela B B, et al. Writing productive stencil codes with overlapped tiling[J]. Concurrency and Computation: Practice and Experience, 2009, 21(1): 25-39.

[3]: Zhou X, Giacalone J P, Garzarán M J, et al. Hierarchical overlapped tiling[C] //Proceedings of the Tenth International Symposium on Code Generation and Optimization. 2012: 207-218.