Case study about LULESH challenge program

Team member: NanMu, RuijieWang

Abstract

In this project, our work mainly about four parts, firstly, we analyzed the development of LULESH program. Then adjusting make file to compile each model, running six types of LULESH model (Serial, MPI, OpenMP, Hybrid, CUDA, a tuning OpenMP model), using profile tool get insight these models. Next, we did five group of experiments to test these models' performance with different configuration. Also, we modify the OpenMP model, getting experience with schedule method and test performance. This report will show work and results about each part.

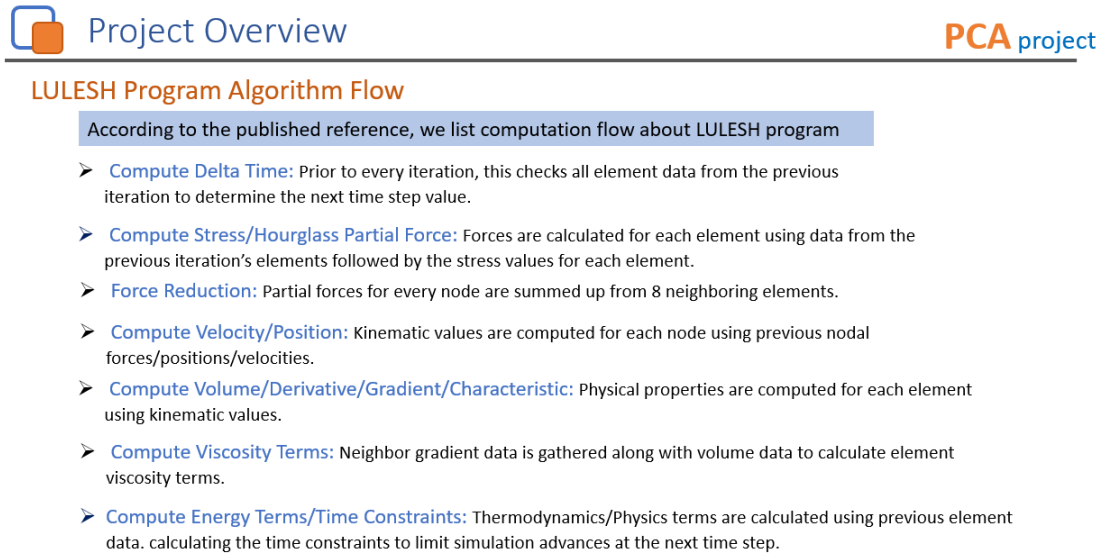
Introduction about LULESH problem

What is about the LULESH

LULESH is a fully featured hydrodynamics mini-app developed by Lawrence Livermore National Laboratory that simulates the effect of a blast wave in a physical domain through explicit time-stepping. Hydrodynamics is a challenging problem and was previously shown to account for a significant fraction (27%) of computing resources used by the Department of Defense. The original LULESH specification is physics code that operates on an unstructured hexahedral mesh with two centering. There is an element centering that stores data on thermodynamic and physical properties and a nodal centering that stores kinematics values such as position and velocity of those points.

LULESH algorithm flow

slide 1 shows the program algorithm flow



Slide 1

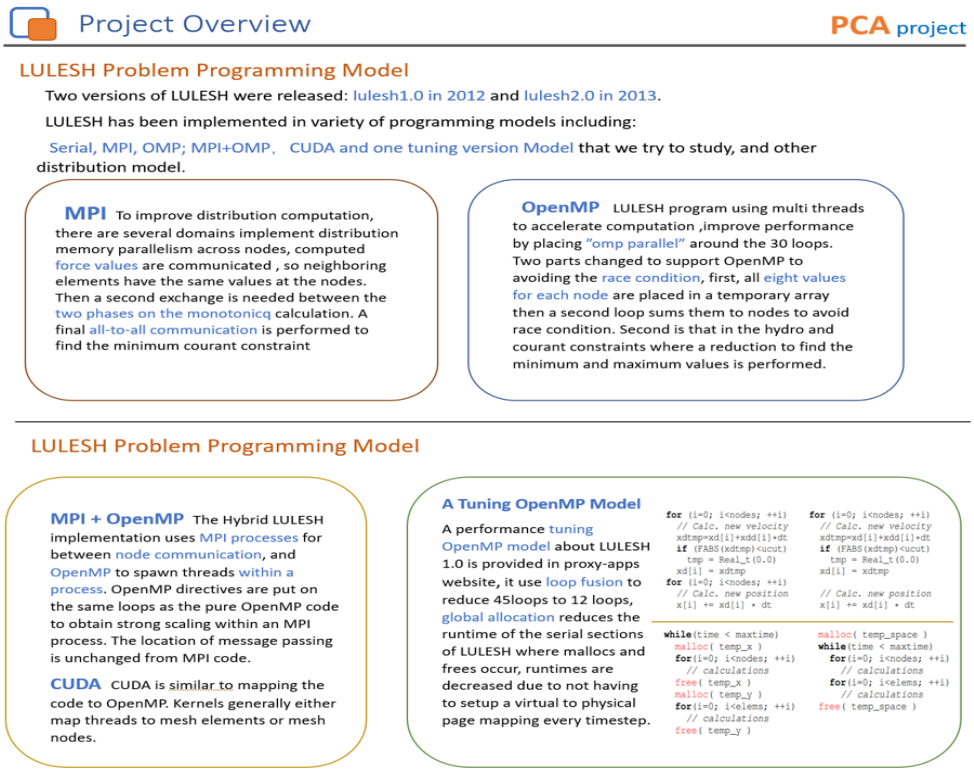
The actual program flow involves a setup and initialization phase where the spatial coordinates of the domain are defined. Then an index set is defined for the single material. The index set is used to mimic multi-material problems and their associated algorithmic costs. Next, the initial problem state and boundary conditions are defined.

According to the published reference, we list computation flow about LULESH program showing on the slide 1

Programming Model

There are two versions of LULESH were released that is LULESH1.0 which released in 2012 and LULESH2.0 which released in 2013. LULESH has been implemented in a wide variety of programming models including: MPI, OMP; MPI+OMP, and other distribution model.

Slide 2 shows the programming model we will study and test



Slide 2

About MPI model, it divides problem workload into each node, and there are three domains need to communicate, one is each rank need to communicate with its neighbor to get the force value. Then the element center value needs to be exchanged between ranks. Finally, the all-to-all are used to collect results and are used for next time stepping. **About OpenMp model**, it mainly "omp for" to divide loop job, there are around thirty loops using ompfor to do parallel calculating. **Hybrid model** using both MPI and OpenMp method. it uses MPI processes for between rank do communication, and OpenMP to distribute threads within a rank. **CUDA model** is similar it is like OpenMp, mainly divide loop work do parallel calculation. **Tuning Model**. The one posted on the web is a tuned Open MP model with some techniques. The loop fusion operating mechanism combining multiple loops over the same iteration space into a single loop. It reduces about 45 loops to 12 loops to reduce the time spent to access the memory. For global allocation, it reduces the times of mallocs and frees to reduce the number of systems calls.

Project Approaches and Results

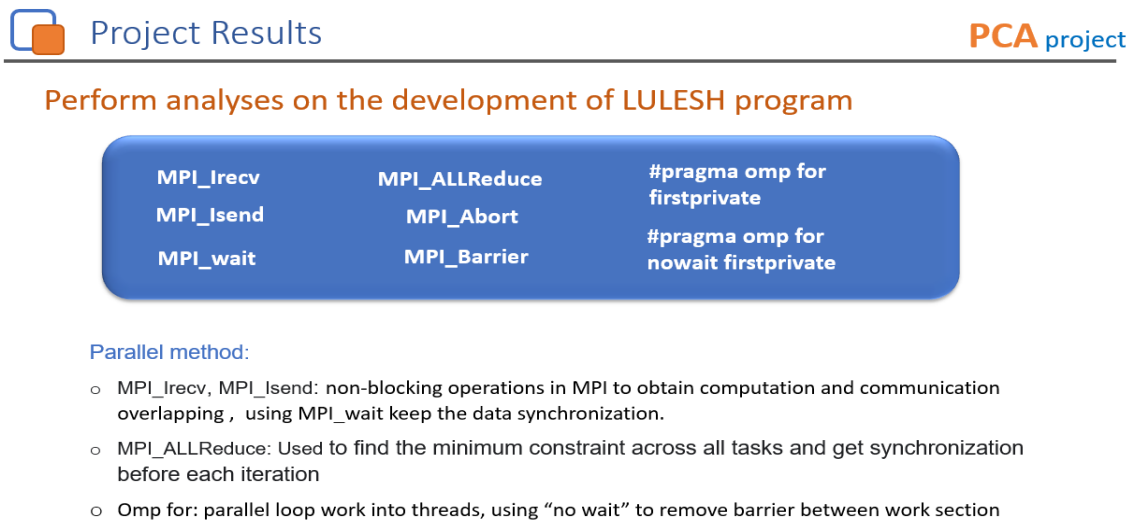
Next part of report is what we did about in this project with results and analysis

Before porting the LULESH Model to HiperGator, we study the serial model structure and comb the distribution instructions that are used in MPI and OpenMP model.

1 Analysis of LULESH distribution method.

Parallel method

Before porting the LULESH Model to Hipergator, we study the LULESH code structure and its development, then we analyze different LULESH model about their distribution method. we comb the MPI and OpenMP instructions which are be used in we find, there are some improvement parallelism methods in the code: first, MPI using non-blocking operations in MPI to be overlapping computation and communication, accompanies with MPI_wait used to acquire parallel improvement and data synchronization. Second, using “no wait” to remove barrier in OpenMP loop.



Slide 3

Below is a list about the parallel instruction using in parallel LULESH model.

1. MPI_Abort: The MPI standard defined a thread-safe interface, but this does not mean that all routines may be called without any thread locks.
2. MPI_Barrier: it blocks all MPI processes in the given communicator until they all call this routine.
3. MPI_ALLReduce are used to letting all ranks get the same results for next calculation
4. MPI_wait: a blocking call that returns only when a specified operation has been completed. It places before the variable that thread currently processing should be used in next section.
5. MPI_Irecv & MPI_Isend: accompanies with MPI_wait to acquire parallel improvement and data symchronization.

6. #Pragma omp for nowait firstprivate: for “firstrpivate”, loop will use the first input value as the initial value. When the for-loop end, this initial value will not be changed. “nowait for “to remove the barrier between work section

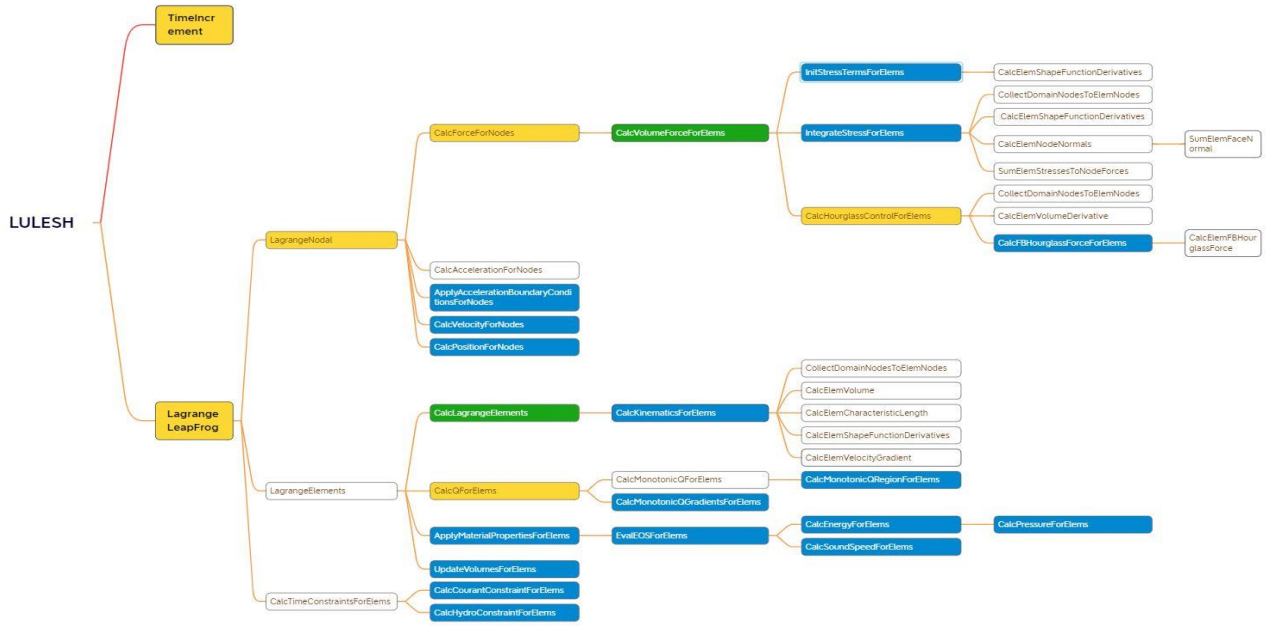
Development of the LULESH code

In this part, we analyze the LULESH structure and draw the structure graph showing in the slide 4.

LULESH program is divided into seven levels, first part is time step calculation, second part is elements and node calculation. upper-level functions use MPI method to divide tasks and scale parallel in node, lower-level functions implement OMP in thread level parallelism. Yellow block means this function using MPI method, blue block means function using OpenMp parallel method, green block shows this function use both of them.

This structure will get more parallelism with less costly communication because it divides tasks at beginning and use many threads parallelism, then communicate between node only in three part. this structure will be good at

the weak scalability when problem size increasing and good at strong scalability when using many threads to distribute computation in a fix problem size.

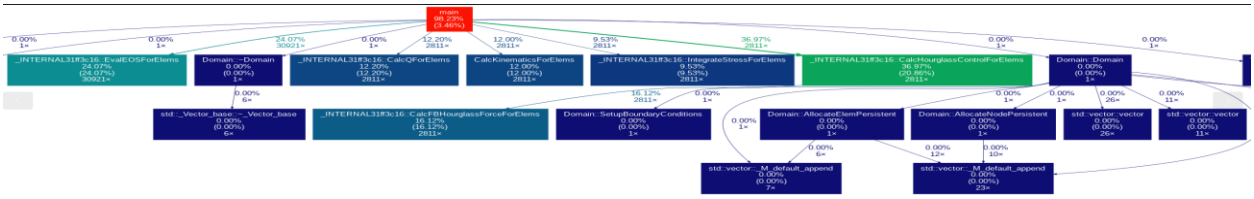


Slide 4

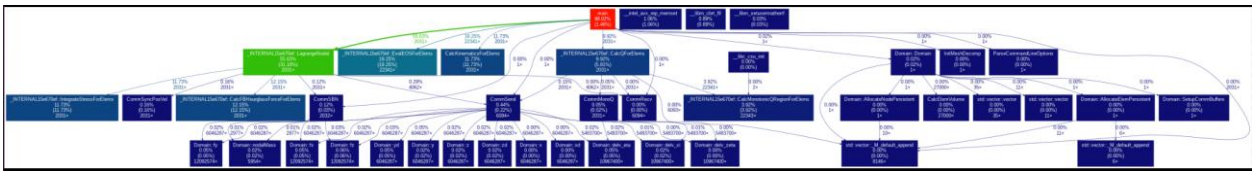
2. Porting, running, and analyzing different LULESH model

In this part, we download serial, MPI, OpenMP and hybrid model to Hipergator, adjust makefile to compile each model code, running these models. then using gprof and gprof2dot tools get insight about each model. This part, we analyze and compare serial, MPI, OpenMp, Hybrid model structure by gprof.

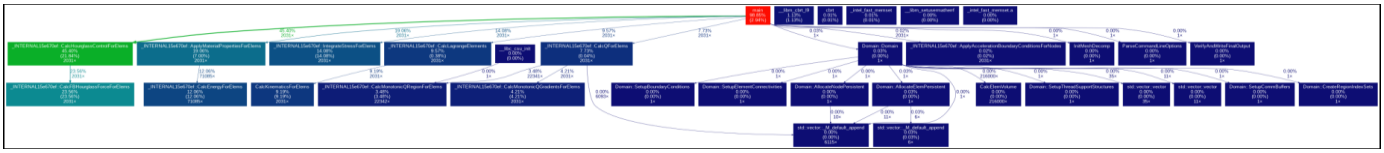
Analyze serial structure: using gprof profile, it shows that function “EvalEOSForElems” take longest time, it takes 24.07% of total time. There are mainly 26 functions to be call, but only 8 of these 26 spend almost 99% running time. the brighter color block represent it takes longer working time.

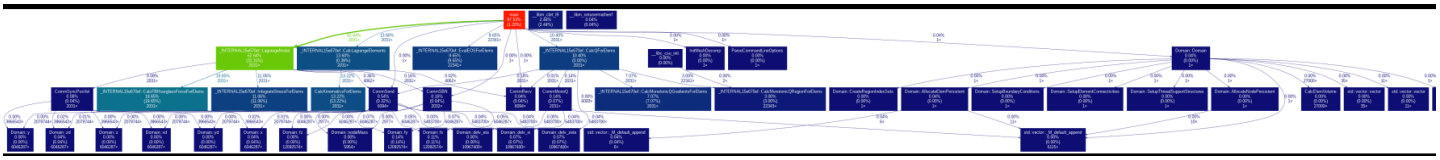


Analyze MPI structure: Compared to serial structure, we find MPI_communication are called to do send receive and synchronization (wait). Their spent total time dosen’t take many running time, it takes running time less than 1 percent. Compared with serial model, many functions which spend long running time such as CalcQForElems, all use MPI parallel divided tasks into ranks. LULESH code using MPI facilitate distributed memory parallelism across nodes.



Analyze OpenMp and Hybrid structure: Compared to serial structure, in Openmp code, top 8 of function all use thread parallelism to do calculation, parallel the “for loop” so that reduce the running time. Hybrid model use both MPI and OpenMP and has similar strategy. Also, these two model a function about “setupThreadSupportStructure” are called to do base setup for the thread parallel.





OpenMp and Hybrid model

Summary and conclusion:

Project results Porting, running and analyzing different LULESH model

No.	function	Ratio of total time	Parallel method	Upper func
1	EvalEOSForElems	24.04%	omp	MPI
2	calcHourglassControlForElems	20.86%	MPI	MPI
3	CalcFBHourglassForceForElems	16.12%	omp	MPI
4	CalcQForElems	12.2%	MPI	MPI
5	CalcKinematicsForElems	12%	omp	MPI
6	IntegrateStressForElems	9.53%	omp	MPI
7	main	3.46%	MPI	

Functions which consuming most time in Serial model

Summary of Parallel Strategy:

- Functions which consuming most running time are similar
- All used parallel method (MPI or OpenMP) to improve performance in different models
- MPI communication mode dose not take too much total time because it using non-block overlapping communication with computation. The data changing only happen in three places which minimal connections among each node
- upper function use MPI divide workload, subfunction use openmp speed up loop work

Slide 5

Here is a summary for these four models About the LULESH Parallel strategy.

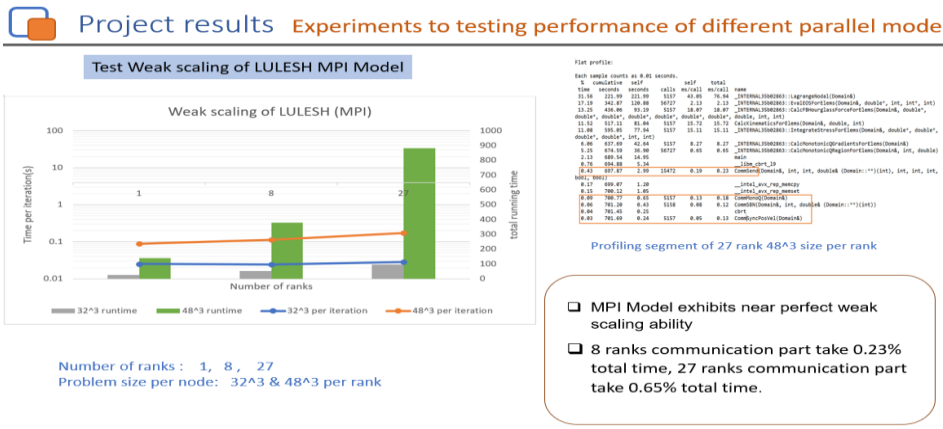
First, we find The LULESH problem use domain decomposition (data parallellism) not Functional Decomposition. Functions which spend most of total time in these four models are similar. Then Look at the chart, it shows that the top 7 functions which take about ninety percent running time in serial code, and these functions have all used parallel method (MPI or OpenMP) to improve performance in different models. Also, MPI_communication section does not take too much total time because it uses non-block instruction to hide the communication.

3. Weak scaling of LULESH MPI Model and MPI+OMP Model

After analyzing the development of the LULESH code, we did five groups of experiments to test each model performance. From this part, we will display the results of these experiments.

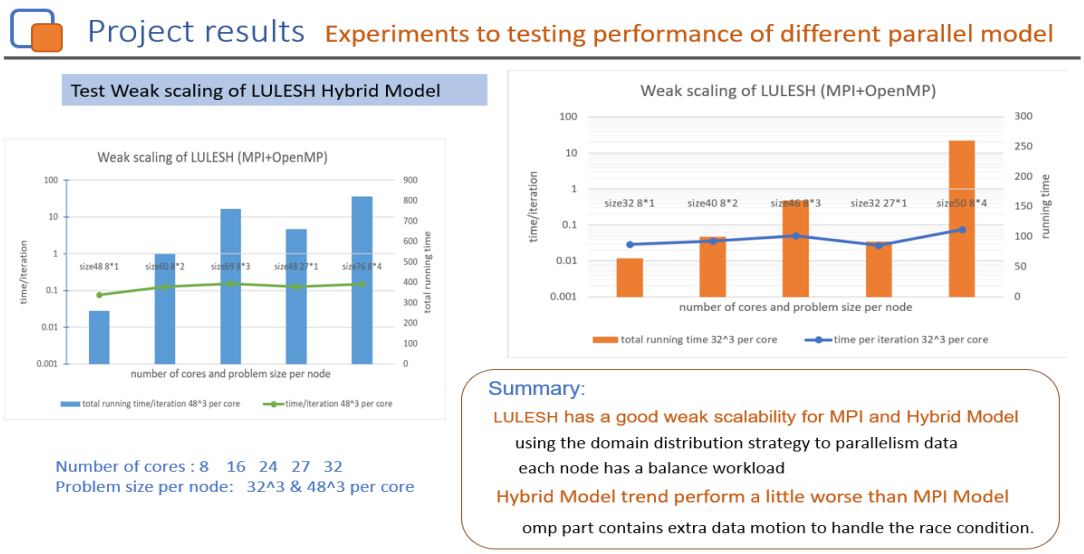
In this part, we test the weak scalability about the MPI Model and Hybrid Model, then analyzing their performance with different configuration.

(1) weak scalability about MPI Model



Slide 6

For MPI Model, there is a restriction when running LULESH: the number of domains, which is equal to the number of MPI tasks, must always be the cube of an integer. This limits us to use only 1, 8 or 27 nodes. So we test these 3 number of node with problem size 32^3 and 48^3 per node. **Analysis:** As shown in the slide 7, with both the number of ranks and the total problem size increase, the MPI version of the code exhibits near perfect weak scaling ability, even though spend longer time to run, but it not proportional increase. since there is not much communication in LULESH, for example, 8 ranks communication part take 0.23% total time, 27 ranks communication part take 0.61% total time, so there should be a good performance when increase the problem size by add more ranks.



Slide 7

(2) weak scalability about MPI+OpenMP Model

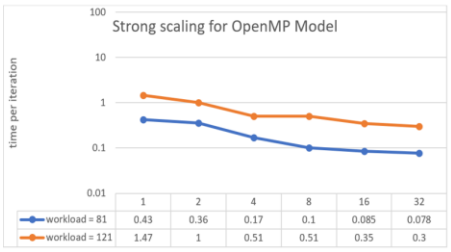
Analysis: For mpi+omp model we change total number of cores (ranks*thread) to test weak scale ability for the MPI+OpenMP model, we keep problem size 32^3 per core, increase total problem size with cores increasing. We find that the hybrid version displays similar scaling characteristics like MPI Model, it also displays a good weak scaling, but a little worse trend of performance than MPI Model. There are multiple reasons. First, compared MPI, Hybrid model need to undertake more workload per rank even though there are more than one cores per rank. Second, the OpenMP code contains extra data motion to handle the race condition when summing to the nodes in the hourglass and stress computations.

Conclusion: LULESH problem perform a perfect weak scalability in both MPI Model and MPI+OpenMP Model, because LULESH problem using the domain distribution strategy to parallelism data, so, each node has a balance workload. It is helpful when increasing the problem size while increasing the number of processors.

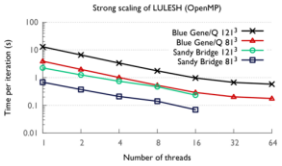
4. Test strong scaling for OpenMP Model

Project results Experiments to testing performance of different parallel model

Test strong scaling for OpenMP Model



Number of cores :1 to 32
Problem size per node: 81^3 & 121^3



Testing result from LULESH research paper

- OpenMP Model perform better than Blue Gene, same as Sandy Bridge,
- Exhibits good strong-scaling behavior in both problem size, indicating that most loop work using thread get good parallelism in the code

Slide 9

This part, we test the strong scalability for Open MP model, changing the cores from 1 to 32 with the problem size 81 and 121. Below is the running time per iteration for each configuration.

Analysis: Slide 9 shows the results, the blue point represents our workload selected as 81 and it show a relatively good strong scaling graph. With the number of threads increase, the time per iteration will decrease. There is a bottleneck around thread's number equals 8, because after the threads increase surpass 8 the decrease rate will drop, and the graph tend to flat. Both two set of problem size test shows the good scalability.

5. Test and analyze a tuning OpenMP Model

In this part, we test the performance of optimization code and compare this allocation +loop fusion 1.0 with original 2.0 in Open MP model, the problem size is 70, changing threads number from 1 to 32. Slide 10 shows the results, time_v1 represents the loop fusion model and time_v2 represents the original LULESH 2.0 model.

Project results Experiments to testing performance of different parallel model

Test the Tuning OpenMP Model

Workload = 70; node = 1; thread = 1					
V1(with loop fusion)			V2(without loop fusion)		
Function	%Time	Time	Function	%Time	Time
CalcFBHourglassForceForElems	37.12	114.60	CalcHourglassControlForElems	19.90	143.77
LagrangeElements	24.90	76.87	CalcEnergyForElems	17.79	128.57
CalcKinematicsForElems	16.41	50.66	CalcFBHourglassForceForElems	15.57	112.54
IntegrateStressForElems	16.22	50.06	ApplyMaterialPropertiesForElems	11.71	84.58

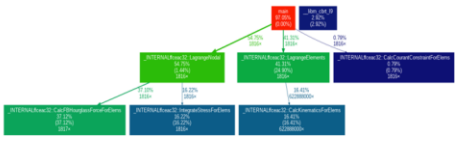


Figure V1 optimization model gprof structure

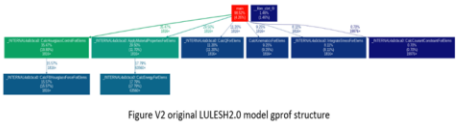
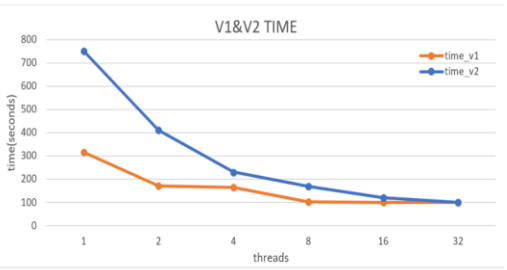


Figure V2 original LULESH2.0 model gprof structure



Summary:

- OpenMP model with loop fusion and global allocation method, perform better than OpenMP with small number cores. Gap vanishing by increase the number of threads
- Functions used loop fusion take larger workload but does not spend more time.
- Be useful for the MPI model which MPI model perform well on weak scalability

Slide 10

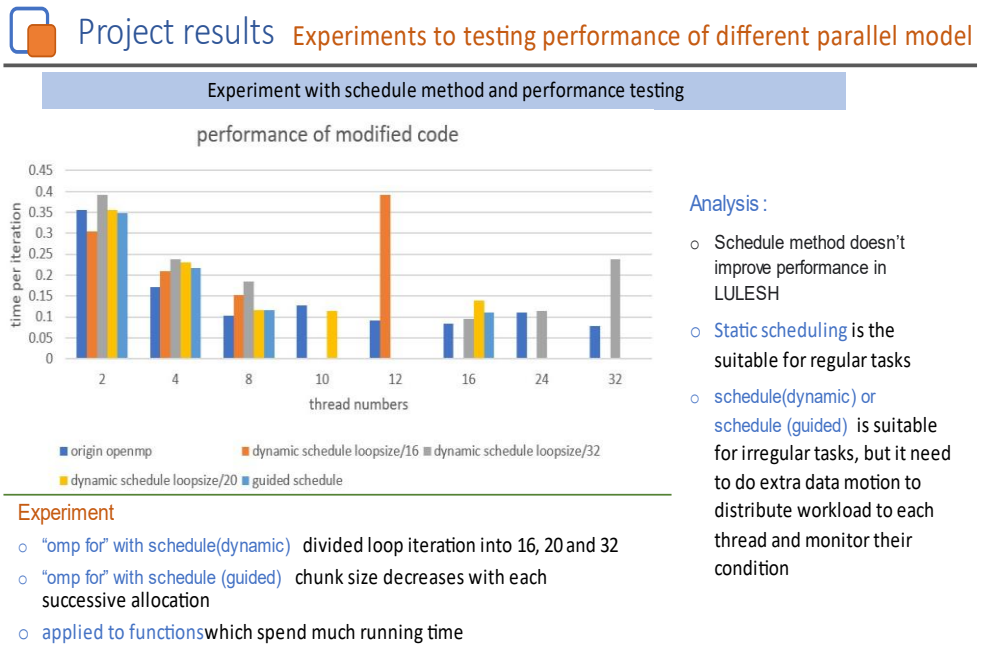
Analysis: According to the comparison of running time, we find if the threads number is small, they have a great gap, loop fusion and global allocation reduce the time spent to access the memory or the number of systems calls. But with the number of threads increasing, they tend to converge with each other. Thus, the loop fusion optimization will be useful for the MPI model which MPI model perform well on weak scalability, and loop fusion will be helpful to reduce the memory access time in each rank, if only consider the OpenMP model, it would not play an important role with the threads number increase.

Then we try to use gprof tool go insight to see the difference between these two models. Table shows the top running time function in each model. Figure shows structure of these two models.

From the code instruction, loop fusion strategy is used in function "CalcFBHourglassForceForElems", from the table, we find this function take lots of time in both version 1 and version 2 code, "CalcFBH" function in version 1 undertake more works within one loop, occupy a greater percentage of time than version 2. However, it takes almost the same amount of time as this function spend in version 2. That means loop fusion method decrease the time spending on the communication and data exchange from cache and memory. Also, we find there are loop fusion and global allocation method using in other functions in v1 version (blue annotation in chart). According to our analysis, several children functions are fused to other function such as CalcKinematicsForElems and IntegrateStressForElems, which results in a different proportion of time allocation.

6. Experiment with schedule method and performance testing.

In this part, we do some experiment about the Open MP model, trying to change parallel loop schedule from default (static) to dynamic schedule with different chunksize, and to guided schedule. Then, apply to some function which spend much running time such like "CalcQForElems "in original LULESH code. We test four scenarios: guided schedule and dynamic schedule with divided loop size into 16, 20 and 32. Testing performance with problem size 81 and different cores. Slide 11 shows the running time per iteration. Compared structure of schedule and no schedule by gprof, there is no obvious improvement in schedule(dynamic) code.



Analysis: There are two trend we observed, firstly, compared with the original OpenMp model, we see that after changing to dynamic schedule, whatever the chunk size is, the performance become worse than before. Secondly, we observed a trend is in these three-chunk size, there is a trend for speed that first fall then rise by increasing core numbers, that is, when the core numbers equal the half of divided factor, the running time is shortest.

Compared two trends, we try to find the reason why the speed become slow. We think that dynamic schedule may has higher overhead than the static scheduling type because it dynamically distributes the iterations to threads during the runtime that need to do extra data motion and monitor threads condition, these management may increase the total running time. So, using many dynamic schedules will lose some performance. Schedule guided, same as dynamic, but the chunk size decreases with each successive allocation, it has more flexibility to distribute workload but still need extra data motion.

As a result, we have summary about this part experiment, first, static scheduling is the suitable for regular tasks, schedule(dynamic) or schedule (guided) is suitable for irregular tasks, but it need to do extra data motion to distribute workload to each thread and monitor their condition

7. Porting and testing LULESH CUDA Model

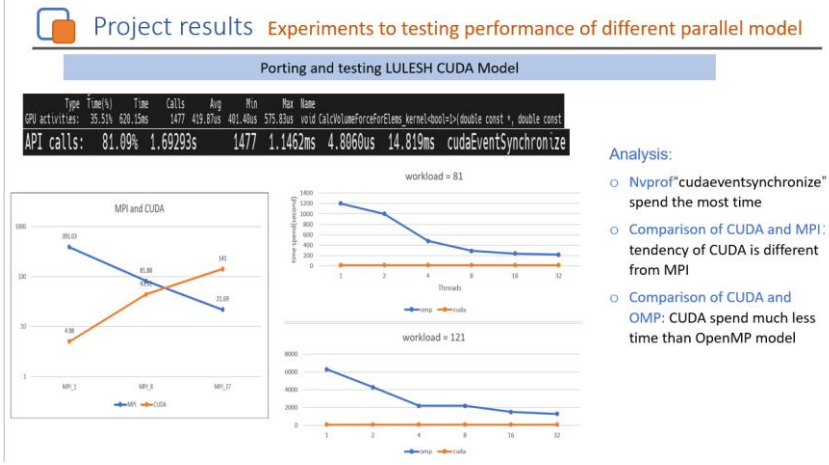
We tried run LULESH CUDA version on HPG. The results come from different workload such as 48,81,121 and we compared these results with what we run previously. Later, we use nvprof to analyze CUDA resource usage.

Nvprof LULESH and analysis

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   35.51%  620.15ms      1477  419.87us  401.40us  575.83us  void CalcVolumeForceForElems_kernel<bool=1>(double const *, double const
```

```
 API calls:   81.09%  1.69293s      1477  1.1462ms  4.8060us  14.819ms  cudaEventSynchronize
```

These two snipping images is the main point of our nvprof result. We can see that cudaeventsynchronize spend the most time. The reason of which we analyze is because each thread need synchronize processing or it will go dead. From the first image, Calcvolumeforcefr elems occupy most of time, it may produce some result that were used by some other functions.



Slide 12

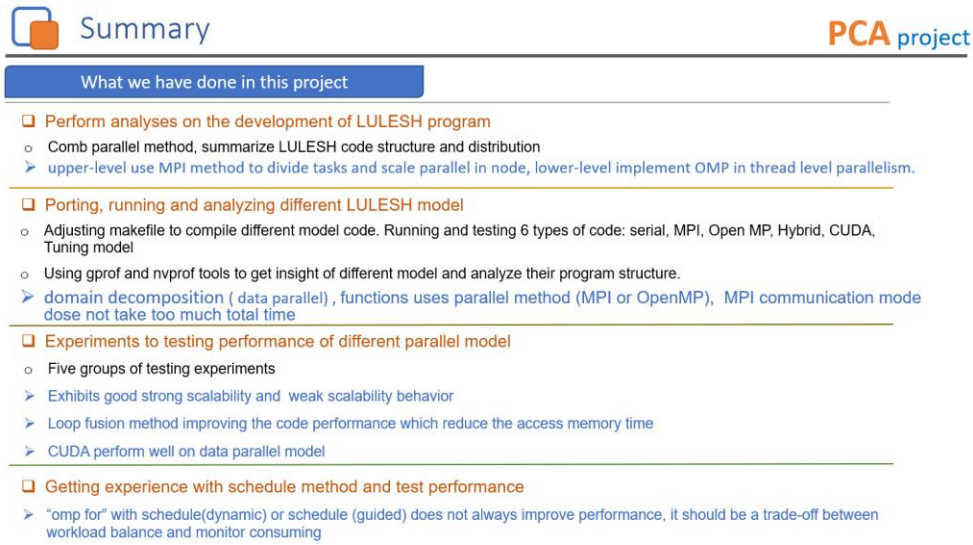
Comparison of CUDA and MPI GPU: NVIDIA GeForce RTX 2080 Ti Problem size = 60;

We can see the tendency of CUDA is different from MPI. It may because in CUDA program, each thread will execute a whole program, it will not divide program into several pieces and do parallel. We have a hypothesis that this program can be divided, but when LULESH program map to CUDA format, they did not create an algorithm to reasonable connect MPI and CUDA.

Comparison of CUDA and OMP GPU: NVIDIA GeForce RTX 2080 Ti Problem size: 81

In fact, in CUDA test, we cannot set a specific thread number for CUDA program, our hypothesis is that when using GPU to execute a program, it will spend t heir whole resource to process currant task which is also correspond to scalability concept. According to trend figure, we can see that program executed with CUDA spend much less time than omp. We analyze that is may because GPU contain greater threads than we thought, thus every thread can execute a small part of program which accelerated this whole process.

Conclusion and summary



Slide 13

Slide 13 shows the summary about what we have done in this project. We focus on four parts of work, and the blue line on slide 13 is analysis conclusion for what we get from this project.

1. We analyze the development of LULESH program. summarize LULESH code structure and function distribution. Upper-level use MPI to divided tasks lower-level implement OMP in thread level parallelism.
2. We porting LULESH code to hypergator and adjust makefile make them can compile in our workspace. We testify 6 types model, After running, we used gprof and nvprof tools to get insight of different model and analyze their program structure. Our conclusion is that it applies data parallel with mpi and omp model. MPI communication mode does not take too much time.
3. Furthermore, we run this code with different configuration to get bunch of data and we analyze them. we did strong and weak scaling, it exhibits good strong and weak scalability. We compared loop fusion version with normal version code. Loop fusion one performance better because it reduces access memory time. We also did the cuda experiment, we learned how to compile cuda and it refresh our understanding of its performance.
4. we also modify OpenMP model to get experience with schedule method, we change the "omp for" schedule(default) to "omp for" with schedule(dynamic) or schedule (guided) but we find it changes does not always improve performance. it is a tradeoff between workload and data monitor.

Reference List

[1]. HYDRODYNAMICS CHALLENGE PROBLEM R. D. Hornung, J. A. Keasler, M. B. Gokhale July 5, 2011

[2] LULESH 2.0 Updates and Changes I. Karlin, J. Keasler, J. R. Neely August 6, 2013

[3] LULESH Programming Model and Performance Ports Overview I. Karlin December 21, 2012

[4] Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application Ian Karlin† , Abhinav Bhatele† , Jeff Keasler† , Bradford L. Chamberlain‡‡, Jonathan Cohen† , Zachary DeVito¶, Riyaz Haque‡ , Dan Laney† , Edward Luke∗ , Felix Wang§ , David Richards† , Martin Schulz† , Charles H. Still†

[5] Optimizing Explicit Hydrodynamics for Power, Energy, and Performance E. A. Leon, I. Karlin, R. E. Grant April 23, 2014, International Conference on Cluster Computing Chicago, IL, United States September 8, 2015 through September 11, 2015

[6] Optimizing the LULESH stencil code using concurrent collections. Chenyang Liu and Milind Kulkarni. 2015. In Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing