HPC 大作业

刘涛

December 23, 2024

1 第一题

1.1 编译CloverLeaf

1.Serial 版本的CloverLeaf
观察Serial文件下的README.MD文件可以知道使用

```
make COMPILER=GNU MPI_COMPILER=gfortran C_MPI_COMPILER=gcc
```

编译运行指令，如下所示：

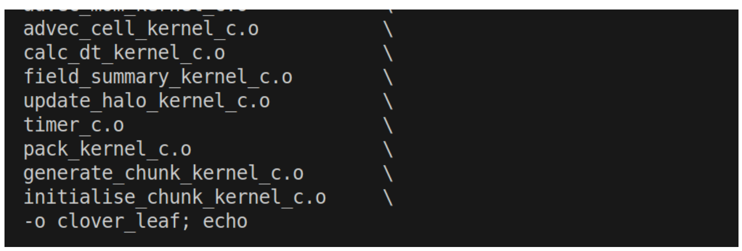


Figure 1: Serial 版本编译结果

直接运行的结果，如下所示:

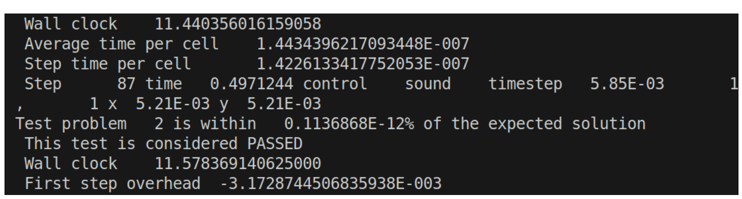


Figure 2: Serial 版本运行结果

2.MPI 版本的CloverLeaf 同理，观察README.MD文件,由于我使用的是GNU编译器，可知采用

```
make COMPILER=GNU MPI_COMPILER=mpifort C_MPI_COMPILER=mpicc DEBUG=1 IEEE=1
```

编译运行指令，如下所示：

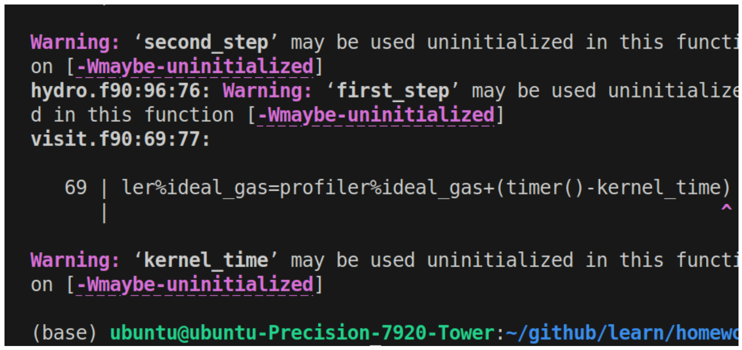


Figure 3: MPI 版本编译结果

直接运行的结果，如图片4所示:

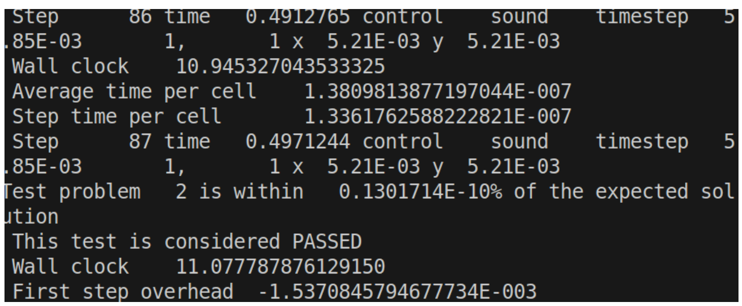


Figure 4: MPI 版本运行结果

3. OPENMP版本的ColverLeaf
继续观察openmp的CloverLeafOpenmp下README.md,同样得到采用相同的编译指令运行OPENMP

make COMPILER=GNU MPI_COMPILER=mpifort C_MPI_COMPILER=mpicc DEBUG=1 IEEE=1

编译运行指令，如图片5所示：

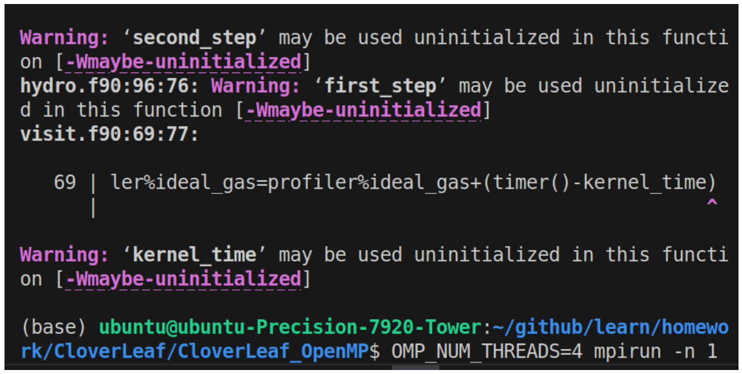


Figure 5: OPENMP 版本编译结果

直接运行的界面，如图片6所示

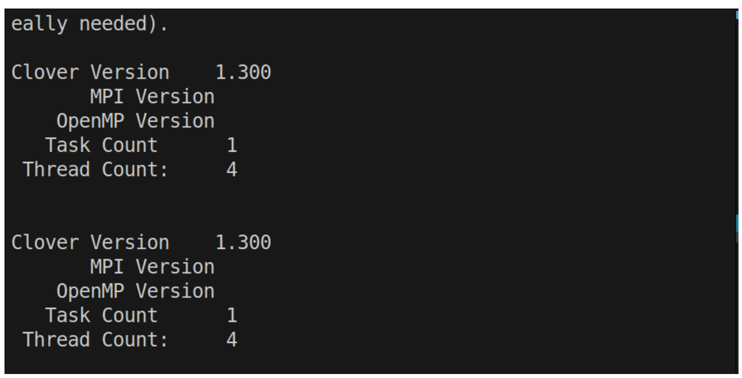


Figure 6: OPENMP 版本运行

运行结果，如图片7所示

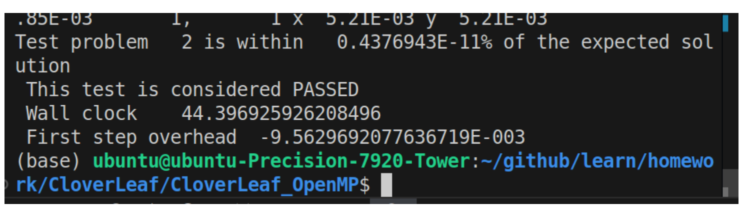


Figure 7: OPENMP 版本运行结果

4. MPI+OPENMP版本的ColverLeaf 到这里，结合已经学习的知识和做过的实验，不难验证编译指令如下所示。

make COMPILER=GNU MPI_COMPILER=mpifort C_MPI_COMPILER=mpicc DEBUG=1 IEEE=1

```
OMP_NUM_THREAD=4 mpirun -n 4 ./clover_leaf
```

测试结果如图片8所示:

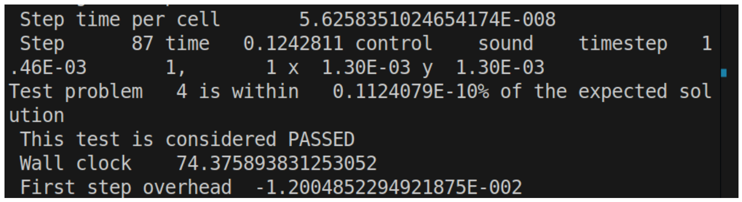


Figure 8: MPI+openmp 版本运行结果

1.2 性能分析

性能分析工具采用intel OneAPI vtune进行处理，这是一款性能优良适合分析的工具。
1.Serial 版本的CloverLeaf性能分析
使用性能分析参数

```
OMP_NUM_THREAD=1 mpirun -n 1 ./clover_leaf
```

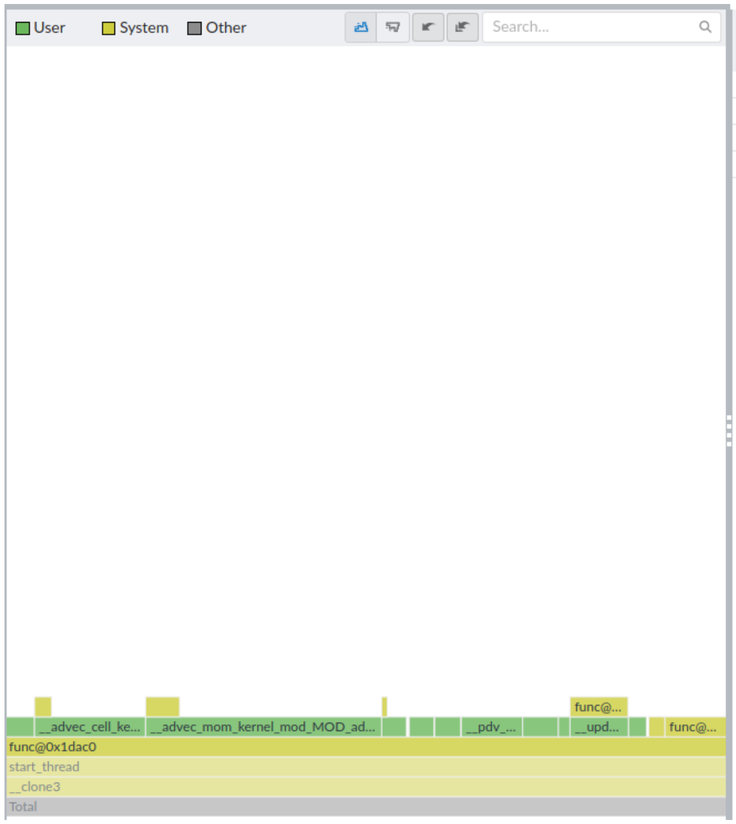
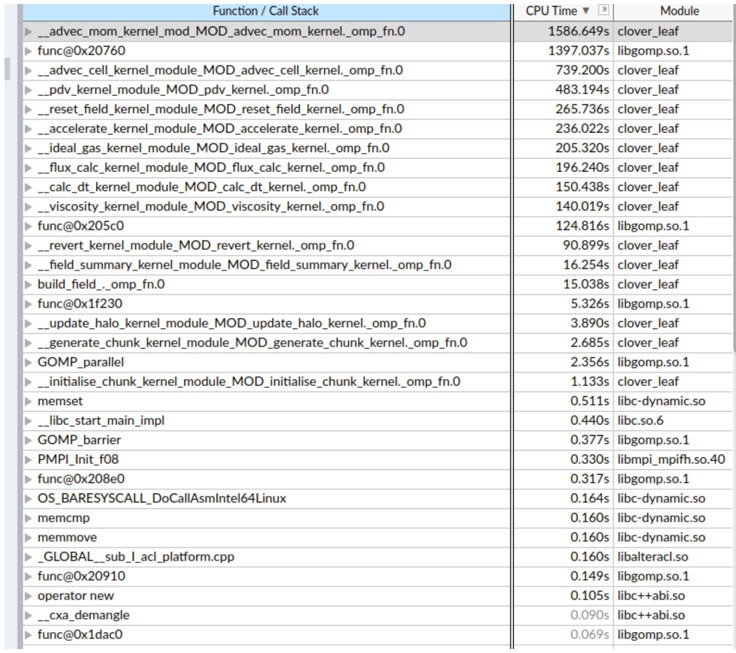


Figure 9: Serial 版本运行火焰图

采用Bottomup的分析模式可以知道程序中的热点位置在于



Function / Call Stack	CPU Time	Module
__advec_mom_kernel_mod_MOD_advec_mom_kernel._omp_fn.0	1586.649s	clover_leaf
func@0x20760	1397.037s	libgomp.so.1
__advec_cell_kernel_module_MOD_advec_cell_kernel._omp_fn.0	739.200s	clover_leaf
__pdv_kernel_module_MOD_pdv_kernel._omp_fn.0	483.194s	clover_leaf
__reset_field_kernel_module_MOD_reset_field_kernel._omp_fn.0	265.736s	clover_leaf
__accelerate_kernel_module_MOD_accelerate_kernel._omp_fn.0	236.022s	clover_leaf
__ideal_gas_kernel_module_MOD_ideal_gas_kernel._omp_fn.0	205.320s	clover_leaf
__flux_calc_kernel_module_MOD_flux_calc_kernel._omp_fn.0	196.240s	clover_leaf
__calc_dt_kernel_module_MOD_calc_dt_kernel._omp_fn.0	150.438s	clover_leaf
__viscosity_kernel_module_MOD_viscosity_kernel._omp_fn.0	140.019s	clover_leaf
func@0x205c0	124.816s	libgomp.so.1
__revert_kernel_module_MOD_revert_kernel._omp_fn.0	90.899s	clover_leaf
__field_summary_kernel_module_MOD_field_summary_kernel._omp_fn.0	16.254s	clover_leaf
build_field_._omp_fn.0	15.038s	clover_leaf
func@0x1f230	5.326s	libgomp.so.1
__update_halo_kernel_module_MOD_update_halo_kernel._omp_fn.0	3.890s	clover_leaf
__generate_chunk_kernel_module_MOD_generate_chunk_kernel._omp_fn.0	2.685s	clover_leaf
GOMP_parallel	2.356s	libgomp.so.1
__initialise_chunk_kernel_module_MOD_initialise_chunk_kernel._omp_fn.0	1.133s	clover_leaf
memset	0.511s	libc-dynamic.so
__libc_start_main_impl	0.440s	libc.so.6
GOMP_barrier	0.377s	libgomp.so.1
PMPI_Init_f08	0.330s	libmpi_mpifh.so.40
func@0x208e0	0.317s	libgomp.so.1
OS_BARESYSCALL_DoCallAsmIntel64Linux	0.164s	libc-dynamic.so
memcmp	0.160s	libc-dynamic.so
memmove	0.160s	libc-dynamic.so
_GLOBAL__sub_I_acl_platform.cpp	0.160s	libalteracl.so
func@0x20910	0.149s	libgomp.so.1
operator new	0.105s	libc++abi.so
__cxa_demangle	0.090s	libc++abi.so
func@0x1dac0	0.069s	libgomp.so.1

Figure 10: Serial 版本运行热点图

由于Serial版本的代码不支持重定位到具体的代码细节，详细的分析在后续写下.

2.MPI 版本的CloverLeaf性能分析 采用如下版本进行分析

```
OMP_NUM_THREAD=1 mpirun -n 4 ./clover_leaf
```

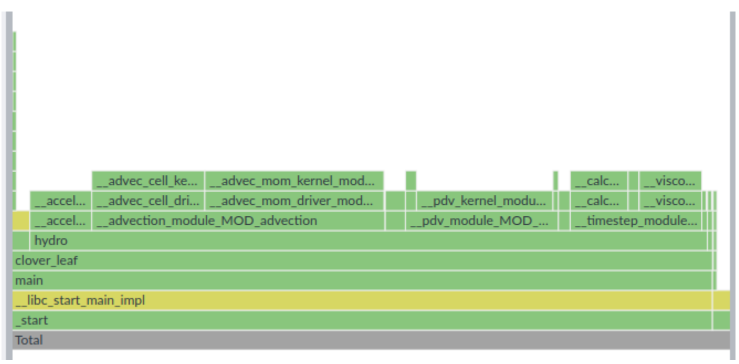


Figure 11: MPI 版本运行火焰图

采用Bottomup的分析模式可以知道程序中的热点位置在于

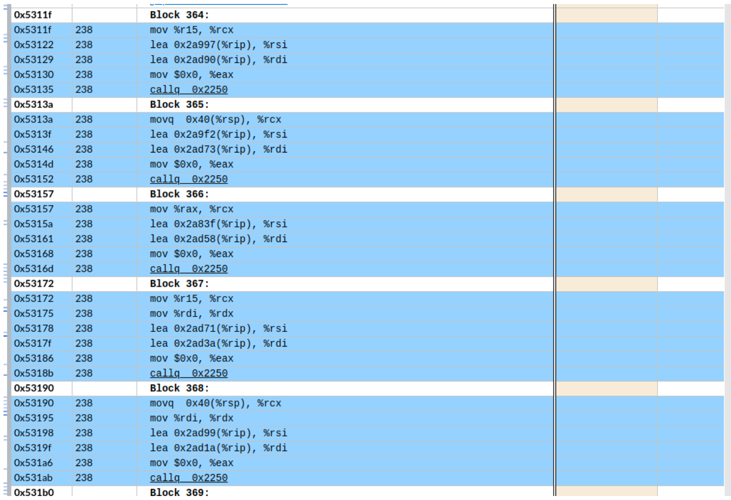
Function / Call Stack	CPU Time ▼	Modul
__advec_mom_kernel_mod_MOD_advec_mom_kernel	9.921s	clover_leaf
__pdv_kernel_module_MOD_pdv_kernel	7.561s	clover_leaf
__advec_cell_kernel_module_MOD_advec_cell_kernel	6.260s	clover_leaf
__viscosity_kernel_module_MOD_viscosity_kernel	3.470s	clover_leaf
__accelerate_kernel_module_MOD_accelerate_kernel	3.429s	clover_leaf
__calc_dt_kernel_module_MOD_calc_dt_kernel	3.210s	clover_leaf
__ideal_gas_kernel_module_MOD_ideal_gas_kernel	1.280s	clover_leaf
__flux_calc_kernel_module_MOD_flux_calc_kernel	1.119s	clover_leaf
__reset_field_kernel_module_MOD_reset_field_kernel	0.650s	clover_leaf
memset	0.482s	libc-dynamic.
__revert_kernel_module_MOD_revert_kernel	0.300s	clover_leaf
__libc_start_main_impl	0.299s	libc.so.6
PMPI_Init_f08	0.298s	libmpi_mpifh.
__field_summary_kernel_module_MOD_field_summary_kernel	0.270s	clover_leaf
build_field	0.220s	clover_leaf
operator new	0.190s	libc++abi.so
OS_BARESYSCALL_DoCallAsmIntel64Linux	0.162s	libc-dynamic.
_GLOBAL__sub_I_acl_platform.cpp	0.120s	libalteracl.so
memmove	0.120s	libc-dynamic.
memcmp	0.110s	libc-dynamic.
__update_halo_kernel_module_MOD_update_halo_kernel	0.080s	clover_leaf
__cxa_demangle	0.060s	libc++abi.so
[Unknown stack frame(s)]	0.040s	
strlen	0.040s	libc-dynamic.
tbb::detail::r1::__TBB_InitOnce::__TBB_InitOnce	0.029s	libtbb.so.12
__memset_evex_unaligned_erms	0.020s	libc.so.6
__svfscanf	0.010s	libc-dynamic.
AllocateNewTlsPerThreadFunc	0.010s	libc-dynamic.
func@0xe4a0	0.010s	libc++abi.so
__initialise_chunk_kernel_module_MOD_initialise_chunk_kernel	0.010s	clover_leaf
_dwarf_skip_leb128	0.010s	libdwarf.so
__tcf_0	0.010s	libalteracl.so

Figure 12: MPI 版本运行热点图

对于在Bottomup图中的消耗CPU时间最多的函数,其代码如下所示

`sigma=ABS(node_flux(j,k))/(node_mass_pre(j,donor))`	0.5%	190.016ms
`width=celldy(k)`	0.1%	49.999ms
`vdiffuw=vel1(j,donor)-vel1(j,upwind)`	0.5%	190.970ms
`vdiffdw=vel1(j,downwind)-vel1(j,donor)`	0.2%	89.996ms
`limiter=0.0`		
`IF(vdiffuw*vdiffdw.GT.0.0)THEN`	0.2%	69.989ms
`auw=ABS(vdiffuw)`		
`adw=ABS(vdiffdw)`		
`wind=1.0_8`		
`IF(vdiffdw.LE.0.0) wind=-1.0_8`		
`limiter=wind*MIN(width*((2.0_8-sigma)*adw/width+(1.0_8+sigma)*auw/celldy(dif))/6.0_8,auw,adw)`	0.1%	50.007ms
`ENDIF`		
`advec_vel_s=vel1(j,donor)+(1.0_8-sigma)*limiter`	1.2%	489.976ms
`mom_flux(j,k)=advec_vel_s*node_flux(j,k)`	1.3%	522.013ms
`ENDDO`		
`ENDDO`		
`DO k=y_min,y_max+1`		
`DO j=x_min,x_max+1`	0.2%	60.008ms
`vel1 (j,k)=(vel1(j,k)*node_mass_pre(j,k)+mom_flux(j,k-1)-mom_flux(j,k))/node_mass_post(j,k)`	1.9%	770.026ms
`ENDDO`		

Figure 13: 源码所在函数



0x5311f		**Block 364:**
0x5311f	238	mov %r15, %rcx
0x53122	238	lea 0x2a997(%rip), %rsi
0x53129	238	lea 0x2ad90(%rip), %rdi
0x53130	238	mov $0x0, %eax
0x53135	238	callq 0x2250
0x5313a		**Block 365:**
0x5313a	238	movq 0x40(%rsp), %rcx
0x5313f	238	lea 0x2a9f2(%rip), %rsi
0x53146	238	lea 0x2ad73(%rip), %rdi
0x5314d	238	mov $0x0, %eax
0x53152	238	callq 0x2250
0x53157		**Block 366:**
0x53157	238	mov %rax, %rcx
0x5315a	238	lea 0x2a83f(%rip), %rsi
0x53161	238	lea 0x2ad58(%rip), %rdi
0x53168	238	mov $0x0, %eax
0x5316d	238	callq 0x2250
0x53172		**Block 367:**
0x53172	238	mov %r15, %rcx
0x53175	238	mov %rdi, %rdx
0x53178	238	lea 0x2ad71(%rip), %rsi
0x5317f	238	lea 0x2ad3a(%rip), %rdi
0x53186	238	mov $0x0, %eax
0x5318b	238	callq 0x2250
0x53190		**Block 368:**
0x53190	238	movq 0x40(%rsp), %rcx
0x53195	238	mov %rdi, %rdx
0x53198	238	lea 0x2ad99(%rip), %rsi
0x5319f	238	lea 0x2ad1a(%rip), %rdi
0x531a6	238	mov $0x0, %eax
0x531ab	238	callq 0x2250
0x531b0		**Block 369:**

Figure 14: 反汇编图片

3.OPENMP 版本的CloverLeaf性能分析

```
OMP_NUM_THREAD=4 mpirun −n 1 ./clover_leaf
```

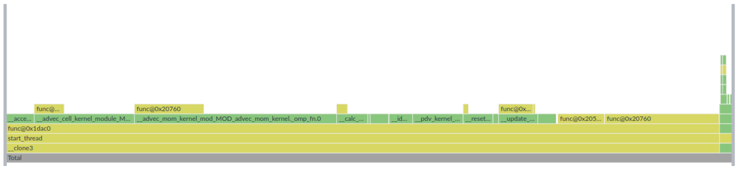
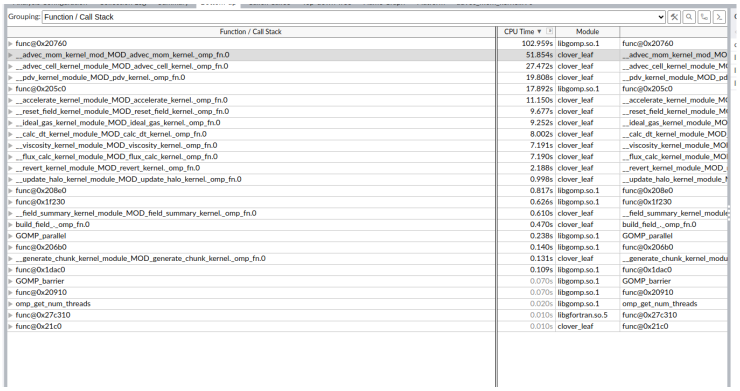


Figure 15: OPENMP 版本运行火焰图

采用Bottomup的分析模式可以知道程序中的热点位置在于

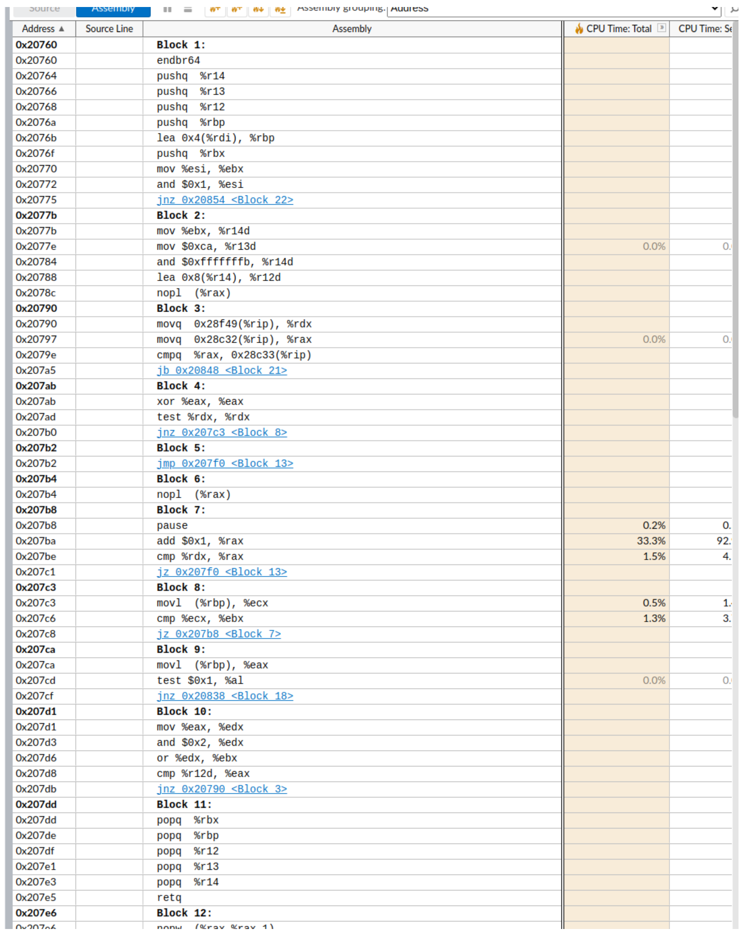


Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Module	
func@0x20760	102.959s	libgomp.so.1	func@0x20760
__advec_mom_kernel_mod_MOD_advec_mom_kernel._omp_fn.0	51.854s	clover_leaf	__advec_mom_kernel_mod_MO
__advec_cell_kernel_module_MOD_advec_cell_kernel._omp_fn.0	27.472s	clover_leaf	__advec_cell_kernel_module_M
__pdv_kernel_module_MOD_pdv_kernel._omp_fn.0	19.808s	clover_leaf	__pdv_kernel_module_MOD_pd
func@0x205c0	17.892s	libgomp.so.1	func@0x205c0
__accelerate_kernel_module_MOD_accelerate_kernel._omp_fn.0	11.150s	clover_leaf	__accelerate_kernel_module_M
__reset_field_kernel_module_MOD_reset_field_kernel._omp_fn.0	9.677s	clover_leaf	__reset_field_kernel_module_M
__ideal_gas_kernel_module_MOD_ideal_gas_kernel._omp_fn.0	9.252s	clover_leaf	__ideal_gas_kernel_module_MO
__calc_dt_kernel_module_MOD_calc_dt_kernel._omp_fn.0	8.002s	clover_leaf	__calc_dt_kernel_module_MOD
__viscosity_kernel_module_MOD_viscosity_kernel._omp_fn.0	7.191s	clover_leaf	__viscosity_kernel_module_MO
__flux_calc_kernel_module_MOD_flux_calc_kernel._omp_fn.0	7.190s	clover_leaf	__flux_calc_kernel_module_MO
__revert_kernel_module_MOD_revert_kernel._omp_fn.0	2.188s	clover_leaf	__revert_kernel_module_MOD_
__update_halo_kernel_module_MOD_update_halo_kernel._omp_fn.0	0.998s	clover_leaf	__update_halo_kernel_module_
func@0x208e0	0.817s	libgomp.so.1	func@0x208e0
func@0x1f230	0.626s	libgomp.so.1	func@0x1f230
__field_summary_kernel_module_MOD_field_summary_kernel._omp_fn.0	0.610s	clover_leaf	__field_summary_kernel_module
build_field_._omp_fn.0	0.470s	clover_leaf	build_field_._omp_fn.0
GOMP_parallel	0.238s	libgomp.so.1	GOMP_parallel
func@0x206b0	0.140s	libgomp.so.1	func@0x206b0
__generate_chunk_kernel_module_MOD_generate_chunk_kernel._omp_fn.0	0.131s	clover_leaf	__generate_chunk_kernel_modu
func@0x1dac0	0.109s	libgomp.so.1	func@0x1dac0
GOMP_barrier	0.070s	libgomp.so.1	GOMP_barrier
func@0x20910	0.070s	libgomp.so.1	func@0x20910
omp_get_num_threads	0.020s	libgomp.so.1	omp_get_num_threads
func@0x27c310	0.010s	libgfortran.so.5	func@0x27c310
func@0x21c0	0.010s	clover_leaf	func@0x21c0

Figure 16: OPENMP 版本运行热点图

此时再进行分析可知出现了系统函数调用func@0x20760,这说明在这个位置调用了OPENMP的系统函数， 反汇编可知这部分的操作没有对应在程序中的代码，在openmp动态链接库文件中被调用,观察其内容,可能 是对应的资源访问互斥锁。

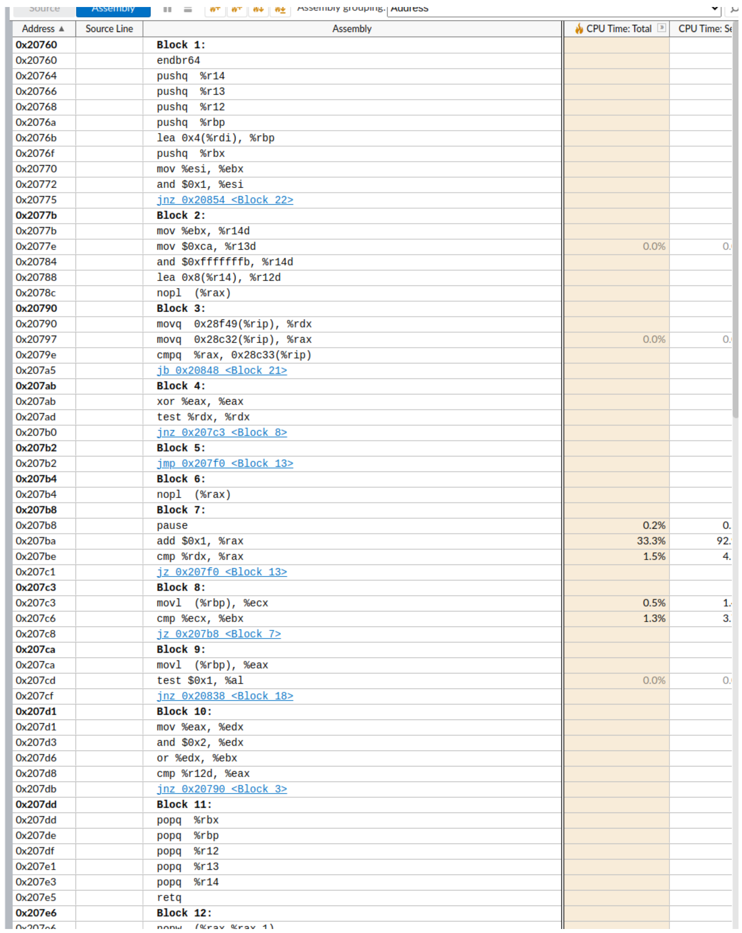


Address ▲	Source Line	Assembly	CPU Time: Total	CPU Time: Se
0x20760		Block 1:		
0x20760		endbr64		
0x20764		pushq %r14		
0x20766		pushq %r13		
0x20768		pushq %r12		
0x2076a		pushq %rbp		
0x2076b		lea 0x4(%rdi), %rbp		
0x2076f		pushq %rbx		
0x20770		mov %esi, %ebx		
0x20772		and $0x1, %esi		
0x20775		jnz 0x20854 <Block 22>		
0x2077b		Block 2:		
0x2077b		mov %ebx, %r14d		
0x2077e		mov $0xca, %r13d	0.0%	0.
0x20784		and $0xfffffffb, %r14d		
0x20788		lea 0x8(%r14), %r12d		
0x2078c		nopl (%rax)		
0x20790		Block 3:		
0x20790		movq 0x28f49(%rip), %rdx		
0x20797		movq 0x28c32(%rip), %rax	0.0%	0.
0x2079e		cmpq %rax, 0x28c33(%rip)		
0x207a5		jb 0x20848 <Block 21>		
0x207ab		Block 4:		
0x207ab		xor %eax, %eax		
0x207ad		test %rdx, %rdx		
0x207b0		jnz 0x207c3 <Block 8>		
0x207b2		Block 5:		
0x207b2		jmp 0x207f0 <Block 13>		
0x207b4		Block 6:		
0x207b4		nopl (%rax)		
0x207b8		Block 7:		
0x207b8		pause	0.2%	0.
0x207ba		add $0x1, %rax	33.3%	92.
0x207be		cmp %rdx, %rax	1.5%	4.
0x207c1		jz 0x207f0 <Block 13>		
0x207c3		Block 8:		
0x207c3		movl (%rbp), %ecx	0.5%	1.
0x207c6		cmp %ecx, %ebx	1.3%	3.
0x207c8		jz 0x207b8 <Block 7>		
0x207ca		Block 9:		
0x207ca		movl (%rbp), %eax		
0x207cd		test $0x1, %al	0.0%	0.
0x207cf		jnz 0x20838 <Block 18>		
0x207d1		Block 10:		
0x207d1		mov %eax, %edx		
0x207d3		and $0x2, %edx		
0x207d6		or %edx, %ebx		
0x207d8		cmp %r12d, %eax		
0x207db		jnz 0x20790 <Block 3>		
0x207dd		Block 11:		
0x207dd		popq %rbx		
0x207de		popq %rbp		
0x207df		popq %r12		
0x207e1		popq %r13		
0x207e3		popq %r14		
0x207e5		retq		
0x207e6		Block 12:		
0x207e6		nopw (%rax,%rax,1)		

Figure 17: OPENMP 版本运行热点图

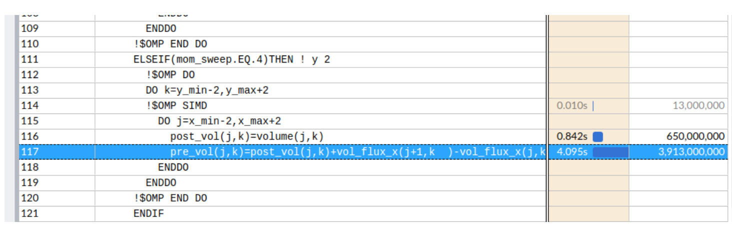
4.MPI+OPENMP 版本的CloverLeaf性能分析

```
OMP_NUM_THREAD=4 mpirun -n 4 ./clover_leaf
```



Address ▲	Source Line	Assembly	CPU Time: Total	CPU Time: Se
0x20760		Block 1:		
0x20760		endbr64		
0x20764		pushq %r14		
0x20766		pushq %r13		
0x20768		pushq %r12		
0x2076a		pushq %rbp		
0x2076b		lea 0x4(%rdi), %rbp		
0x2076f		pushq %rbx		
0x20770		mov %esi, %ebx		
0x20772		and $0x1, %esi		
0x20775		jnz 0x20854 <Block 22>		
0x2077b		Block 2:		
0x2077b		mov %ebx, %r14d		
0x2077e		mov $0xca, %r13d	0.0%	0.
0x20784		and $0xfffffffb, %r14d		
0x20788		lea 0x8(%r14), %r12d		
0x2078c		nopl (%rax)		
0x20790		Block 3:		
0x20790		movq 0x28f49(%rip), %rdx		
0x20797		movq 0x28c32(%rip), %rax	0.0%	0.
0x2079e		cmpq %rax, 0x28c33(%rip)		
0x207a5		jb 0x20848 <Block 21>		
0x207ab		Block 4:		
0x207ab		xor %eax, %eax		
0x207ad		test %rdx, %rdx		
0x207b0		jnz 0x207c3 <Block 8>		
0x207b2		Block 5:		
0x207b2		jmp 0x207f0 <Block 13>		
0x207b4		Block 6:		
0x207b4		nopl (%rax)		
0x207b8		Block 7:		
0x207b8		pause	0.2%	0.
0x207ba		add $0x1, %rax	33.3%	92.
0x207be		cmp %rdx, %rax	1.5%	4.
0x207c1		jz 0x207f0 <Block 13>		
0x207c3		Block 8:		
0x207c3		movl (%rbp), %ecx	0.5%	1.
0x207c6		cmp %ecx, %ebx	1.3%	3.
0x207c8		jz 0x207b8 <Block 7>		
0x207ca		Block 9:		
0x207ca		movl (%rbp), %eax		
0x207cd		test $0x1, %al	0.0%	0.
0x207cf		jnz 0x20838 <Block 18>		
0x207d1		Block 10:		
0x207d1		mov %eax, %edx		
0x207d3		and $0x2, %edx		
0x207d6		or %edx, %ebx		
0x207d8		cmp %r12d, %eax		
0x207db		jnz 0x20790 <Block 3>		
0x207dd		Block 11:		
0x207dd		popq %rbx		
0x207de		popq %rbp		
0x207df		popq %r12		
0x207e1		popq %r13		
0x207e3		popq %r14		
0x207e5		retq		
0x207e6		Block 12:		

Figure 18: OPENMP 版本运行热点图



109	ENDDO		
110	!$OMP END DO		
111	ELSEIF(mom_sweep.EQ.4)THEN ! y 2		
112	!$OMP DO		
113	DO k=y_min-2,y_max+2		
114	!$OMP SIMD	0.010s	13,000,000
115	DO j=x_min-2,x_max+2		
116	post_vol(j,k)=volume(j,k)	0.842s	650,000,000
117	pre_vol(j,k)=post_vol(j,k)+vol_flux_x(j+1,k)-vol_flux_x(j,k	4.095s	3,913,000,000
118	ENDDO		
119	ENDDO		
120	!$OMP END DO		
121	ENDIF		

Figure 19: OPENMP+MPI运行反汇编程序

1.3 效率对比

1.不同MPI+OPENMP的对比 x_cell=960,y_cell=960

OMP_NUM_THREADS	MPI ranks	time consumption
1	1	47.35715s
1	2	24.93825s
1	4	12.38018s
2	1	43.900086s
2	2	22.80418s
2	4	6.75957
4	1	44.56657s
4	2	23.21618s
4	4	4.126149s

Table 1: ref下不同OMP_NUM_THREADS and MPI 的时间消耗

x_cell=1920,y_cell=1920

OMP_NUM_THREADS	MPI ranks	time consumption
1	1	188.6974799s
1	2	96.022246s
1	4	49.126179s
2	2	22.80418s
2	4	26.75957
4	4	19.46512s

Table 2: ref下不同OMP_NUM_THREADS and MPI 的时间消耗

2.不同MPI的对比
xcell=960,ycell=960

OMP_NUM_THREADS	MPI ranks	time consumption
1	1	37.21615s
1	2	19.731973s
1	4	9.932440s
1	8	5.461988s
1	16	3.7412819s

Table 3: ref下不同OMP_NUM_THREADS and MPI 的时间消耗

3.OPEMP版本的对比
xcell=960,ycell=960

OMP_NUM_THREADS	MPI ranks	time consumption
1	1	46.32359s
2	1	24.176760s
4	1	13.139265s
8	1	7.1625220s
16	1	4.6675159s

Table 4: OPENMP下不同OMP_NUM_THREADS and MPI 的时间消耗

1.4 CUDA 编译

由于CUDA版本的编译需要适配,做如下的修改, 使用对应的GNU编译器

```
FLAGS_INTEL      = -O3
FLAGS_SUN        = -O2
FLAGS_GNU        = -O3
FLAGS_CRAY       = -O2 -em -ra -f fre
FLAGS_PGI        = -O2 -Mpreprocess
```

Figure 20: 编译命令修改

由于我的fortran版本的是gcc11对应的fortran版本,所以使用对应的编译器会出现早期编译器对浮点数和int类型坚持不严格,而在新编译器上的检查严格造成的错误.

```
FLAGS=$(FLAGS_$(COMPILER)) $(OMP) $(I3E) $(OPTIONS) -fallow-argument-mismatch
CFLAGS=$(CFLAGS_$(COMPILER)) $(OMP) $(I3E) $(C_OPTIONS) -c
MPI_COMPILER=mpif90
C_MPI_COMPILER=mpicc

NV_FLAGS+=-D MANUALLY_CHOOSE_GPU
```

Figure 21: 编译命令修改

```
FLAGS_INTEL      = -O3
FLAGS_SUN        = -O2
FLAGS_GNU        = -O3
FLAGS_CRAY       = -O2 -em -ra -f fre
FLAGS_PGI        = -O2 -Mpreprocess
```

Figure 22: 编译命令修改

由于我的电脑是Ampere架构的显卡,修改Makefile文件获得计算结果

```
CODE_GEN_MAXWELL=-gencode arch=compute_5
CODE_GEN_PASCAL=-gencode arch=compute_60
CODE_GEN_AMPERE=-gencode arch=compute_86

CUDA_LIB_PATH=/usr/local/cuda-12.4/lib64
LDLIBS+=-lstdc++ -lcudart
```

Figure 23: 修改为Ampere架构

CUDA是12.4类型的CUDA,所以修改对应的CUDA编译器

```
# requires CUDA_HOME to be set - not the
CUDA_HOME=/usr/local/cuda-12.4
NV_FLAGS=-I$(CUDA_HOME)/include $(CODE_G
NV_FLAGS+=-D NO_ERR_CHK
libdir.x86_64 = lib64
libdir.i686   = lib
MACHINE := $(shell uname -m)
libdir = $(libdir.$(MACHINE))
LDFLAGS+=-L$(CUDA_HOME)/$(libdir)
```

```
%.o: %.cu Makefile make.deps
	/usr/local/cuda-12.4/bin/nvcc $(NV_F
%.mod %_module.mod %_leaf_module.mod: %.
	@true
%.o: %.f90 Makefile make.deps
```

Figure 24: 指定CUDA和NVCC位置

采用如下的编译指令

make COMPILER=GNU MPI_COMPILER=mpifort C_MPI_COMPILER=mpicc DEBUG=1 IEEE=1

成功编译对应的程序,其结果如下所示

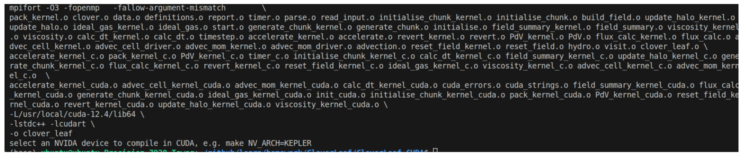


Figure 25: 编译完成

测试结果,如下所示:

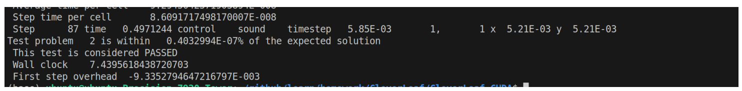


Figure 26: 测试通过

1.5 CUDA版本程序测试

GPU=RTX 3090 CPU=gold 6132 Mpi_num=1 openmp_num=1 可以得出对比

x_cell	y_cell	time consumption
960	960	6.87124s
1920	960	22.105352s
3840	960	38.15225s
1920	1920	27.509817s
3840	1920	54.6727240s
3840	3840	114.67717s

Table 5: CUDA 下不同网格大小的时间消耗

我的GPU和CPU,3090GPU约为15个左右的CPU加速效果

1.6 CUDA代码技巧分析