OFM

Arm crossplatform tools

VI-HPS platform October 16, 2018

© 2017 Arm Limited

An introduction to Arm

Arm is the world's leading semiconductor intellectual property supplier

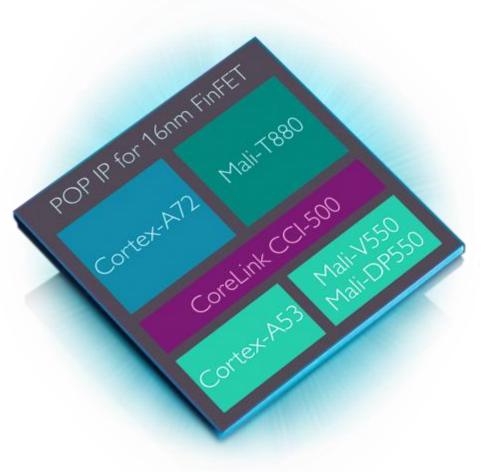
We license to over 350 partners: present in 95% of smart phones, 80% of digital cameras, 35% of all electronic devices, and a total of 60 billion Arm cores have been shipped since 1990

Our CPU business model:

License technology to partners, who use it to create their own system-on-chip (SoC) products

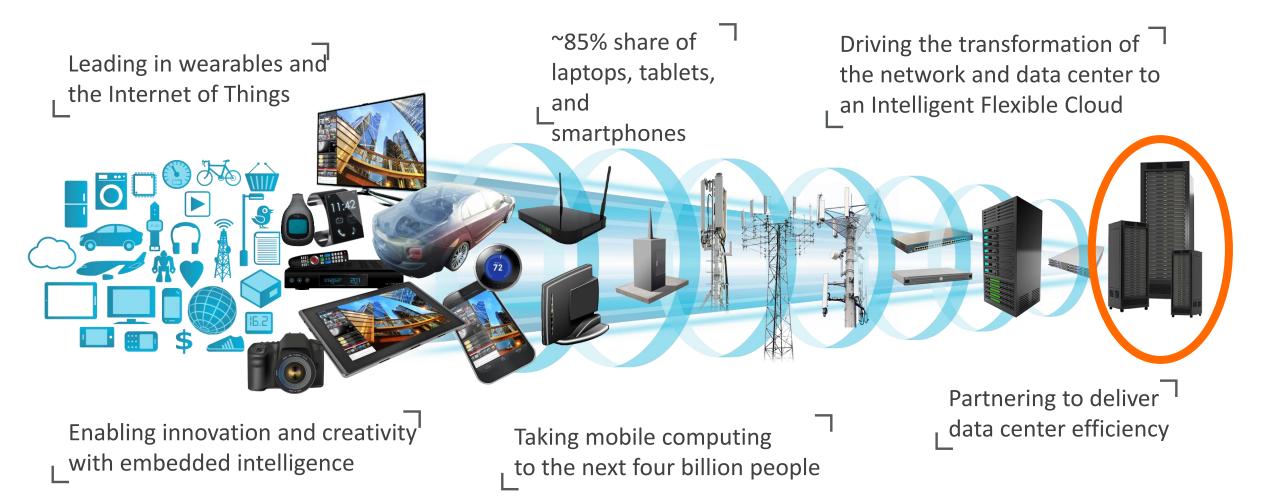
- We may license an instruction set architecture (ISA) such as "Armv8-A"
- or a specific implementation, such as "Cortex-A72"

Partners who license an ISA can create their own implementation, ...and our IP extends beyond the CPU as long as it passes the compliance tests



ARM's mission

Deploy energy-efficient ARM-based technology, wherever computing happens...



HPC strategy



Mission: Enable the world's first Arm supercomputers

Strategy: Enablement + Co-Design + Partnership

Building Blocks

Enablement

- Address gaps in computational capability and data movement within Architecture
- Seed the software ecosystem with open source support for Armv8 and SVE libraries, tools, and optimized workloads
- Provide world class tools for compilation, analysis, and debug at large scale

Co-Design

- Work with key end-customers in DoE, DoD, RIKEN, and EU to design balanced architecture, uArchitecture and SoCs based on real-world workloads, not benchmarks
- Develop simulation and modelling tools to support co-design development with endcustomers, partners, and academia

Partnership

- Work with Architecture partners to bring optimized solutions to market quickly
- Work with ATG & uArchitecture design teams to steer future designs to be more relevant for HPC, HPDA, and ML
- Work with key ISVs to enable mid-market

Arm Allinea Studio



© 2017 Arm Limited

arm Allinea Studio | New commercial bundle

Meets the requirements of HPC developers on Arm



Arm Forge

An interoperable toolkit for debugging and profiling



Commercially supported by Arm





The de-facto standard for HPC development

- Available on the vast majority of the Top500 machines in the world
- Fully supported by Arm on x86, IBM Power, Nvidia GPUs, etc.

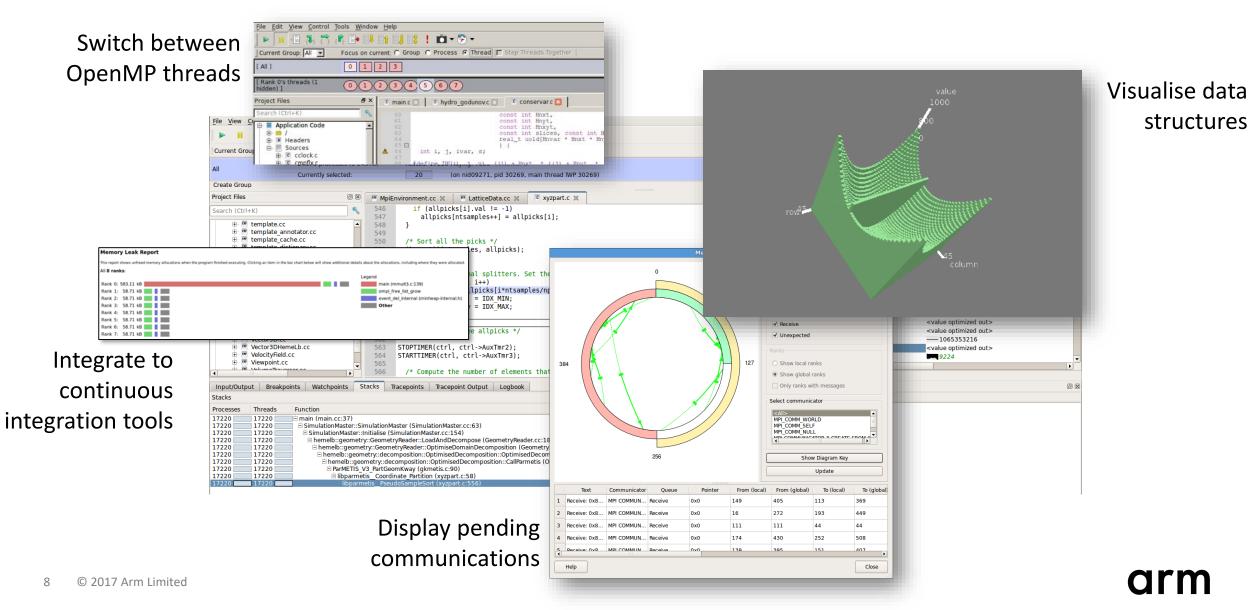
State-of-the art debugging and profiling capabilities

- Powerful and in-depth error detection mechanisms (including memory debugging)
- Sampling-based profiler to identify and understand bottlenecks
- Available at any scale (from serial to petaflopic applications)

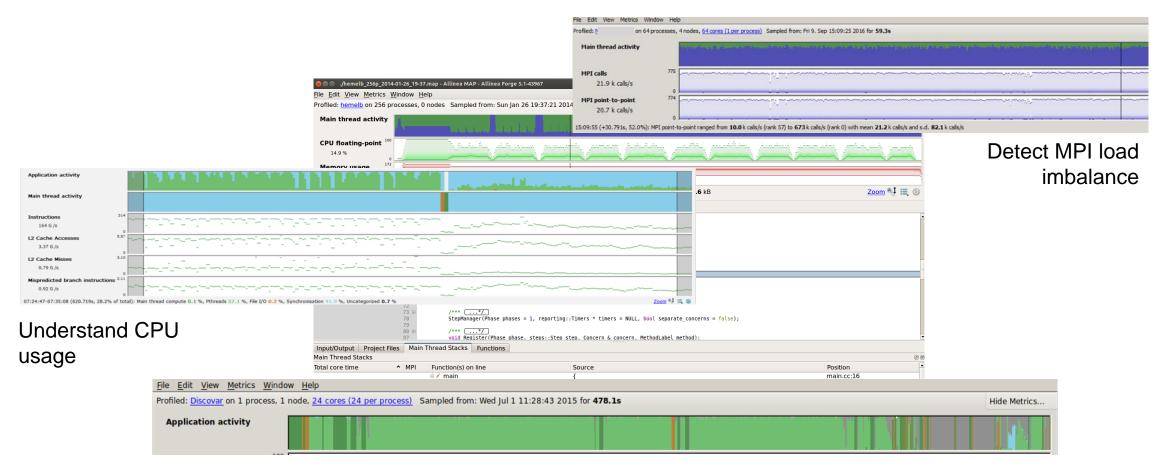
Easy to use by everyone

- Unique capabilities to simplify remote interactive sessions
- Innovative approach to present quintessential information to users





Arm MAP



Identify regions of high OpenMP synchronisation

arm

Arm Performance Reports

 CITM
 Executable:
 MADbench2

 PERFORMANCE
 Resources:
 16 processes, 1 node

 Start time:
 mon Nov 4 12:27:50 2013

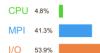
 REPORTS
 Total time:
 109 seconds (2 minutes)

 Full path:
 /tmp/MADbench2

 Notes:
 12-core server / HDD / 16 readers + writers

Summary: MADbench2 is I/O-bound in this configuration

The total wallclock time was spent as follows:



Time spent running application code. High values are usually good. This is **low**; it may be worth improving I/O performance first. Time spent in MPI calls. High values are usually bad.

This is **average**; check the MPI breakdown for advice on reducing it. Time spent in filesystem I/O. High values are usually bad.

This is **high**; check the I/O breakdown section for optimization advice.

This application run was I/O-bound. A breakdown of this time and advice for investigating further is in the I/O section below.

CPU

A breakdown of how the 4.8% total CPU time was spent: Scalar numeric ops 4.9% |

Vector numeric ops 0.1% | Memory accesses 95.0% Other 0.0 |

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance. No time was spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

I/O

A breakdown of how the 53.9% total I/O time was spent:



Most of the time is spent in write operations, which have a very low transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

MPI

Of the 41.3% total time spent in MPI calls: Time in collective calls 100.0% Time in point-to-point calls 0.0% Estimated collective rate 4.07 bytes/s Estimated point-to-point rate 0 bytes/s

All of the time is spent in collective calls with a very low transfer rate. This suggests a significant load imbalance is causing synchronization overhead. You can investigate this further with an MPI profiler.

Memory

Per-process memory usage may also affect scaling:

- Mean process memory usage
 160 Mb

 Peak process memory usage
 173 Mb
- Peak node memory usage 17.2%

The peak node memory usage is low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.

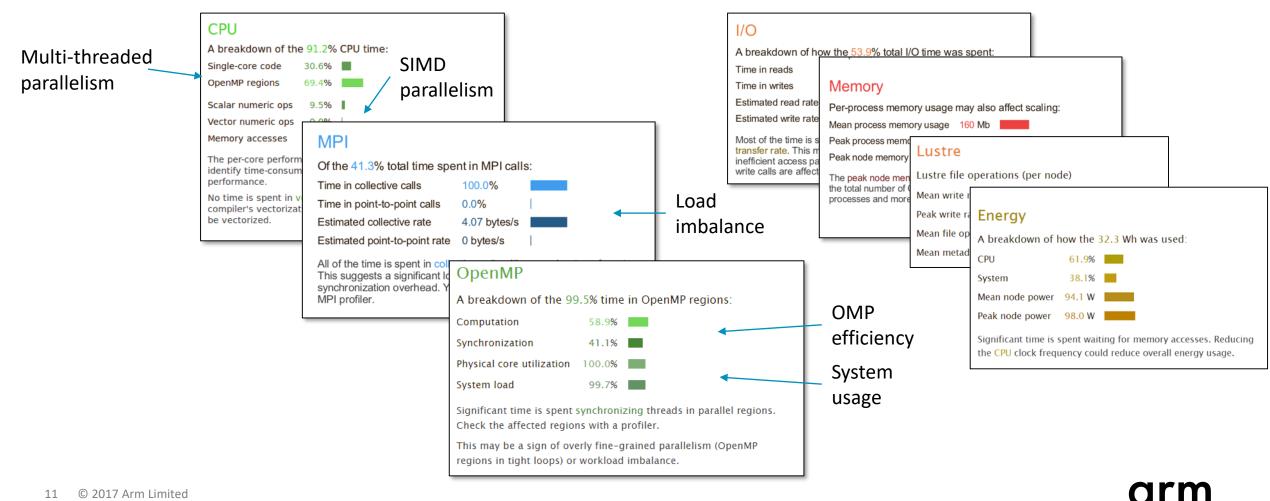
Very simple start-up

Fully scalable, very low overhead

Rich set of metrics

Powerful data analysis

Arm Performance Reports



Imbalance



© 2017 Arm Limited

Challenges

• Workload imbalance results in productivity loss

Longer application
 runtime

- Wasted CPU hours
- Less "science" done per project

End-users

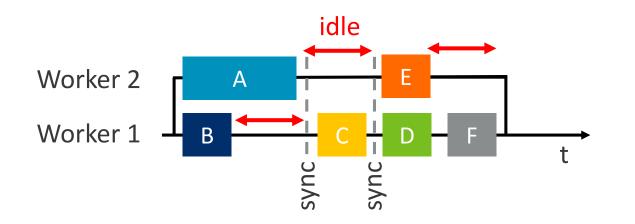
System Administrators

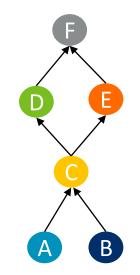
- Pressure to identify & improve inefficient workloads
- Translates the high system usage into a need for more hardware

- Smaller return on investment
- Increased hardware costs to absorb the demand
- Bigger energy bill

System Owners

In practice

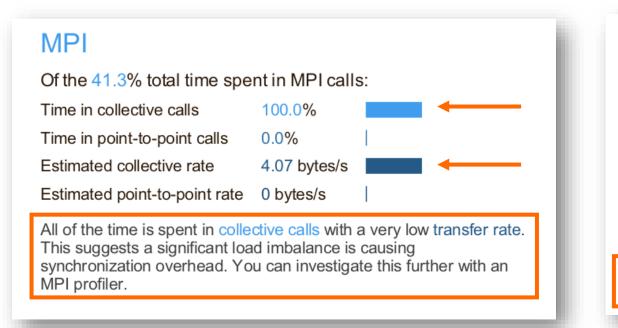




arm

MPI and OpenMP imbalance

- Clues: excessive synchronization
 - MPI collective calls with no actual data transfer
 - Idle cores where threads are stuck in locks/mutexes



OpenMP

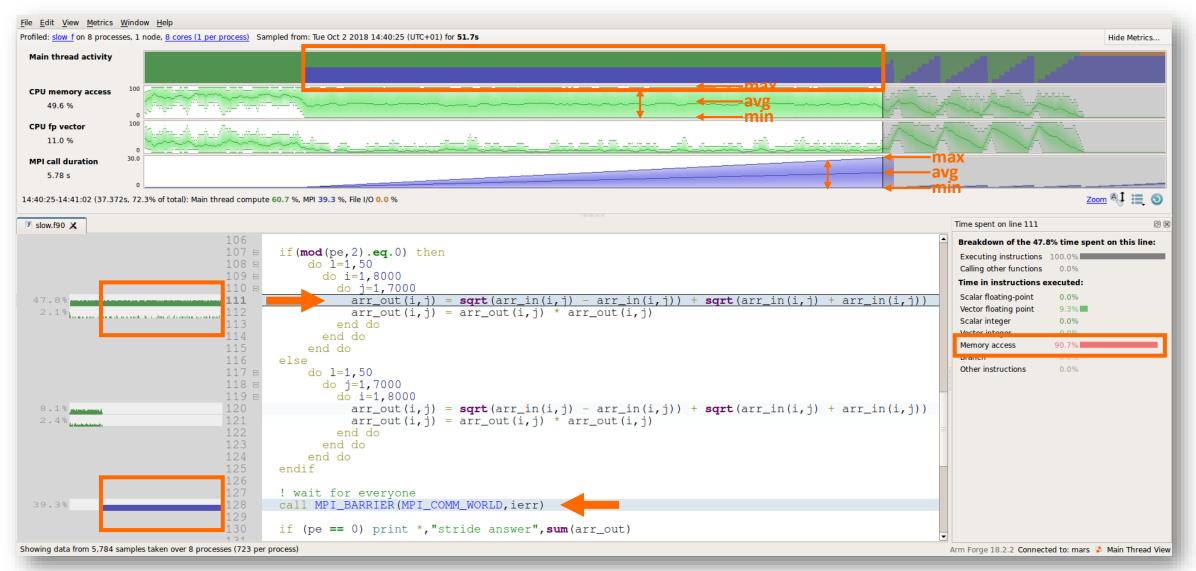
A breakdown of the 74.5% time in OpenMP regions:

Computation	53.6 %	
Synchronization	46.4%	
Physical core utilization	100.0%	
System load	78.0%	

Significant time is spent synchronizing threads in parallel regions. Check the affected regions with a profiler.

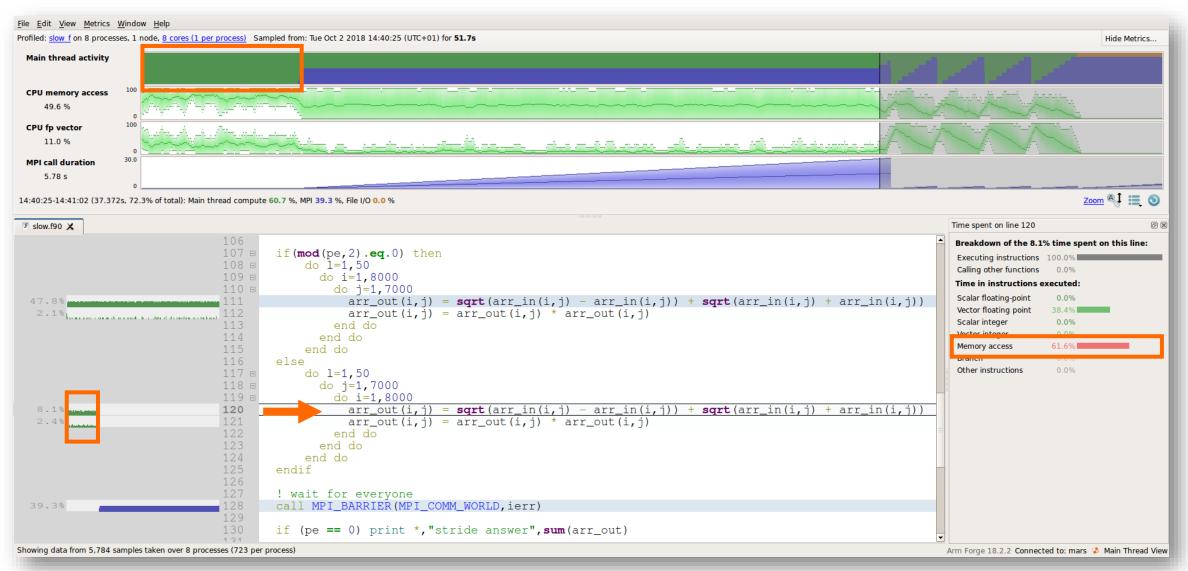
This may be a sign of overly fine-grained parallelism (OpenMP regions in tight loops) or workload imbalance.

MPI imbalance: barrier



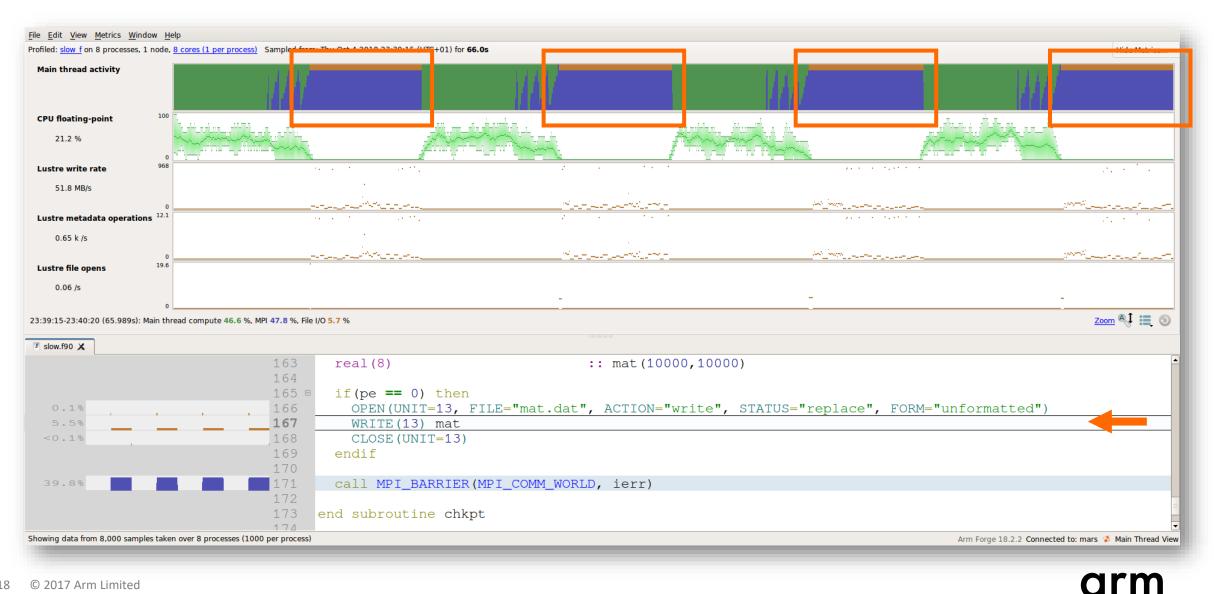
arm

MPI imbalance: barrier

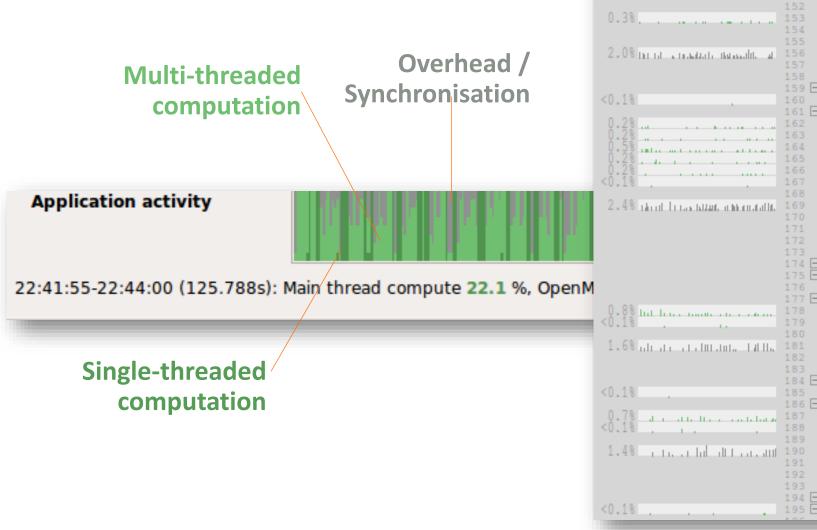


arm

IO imbalance



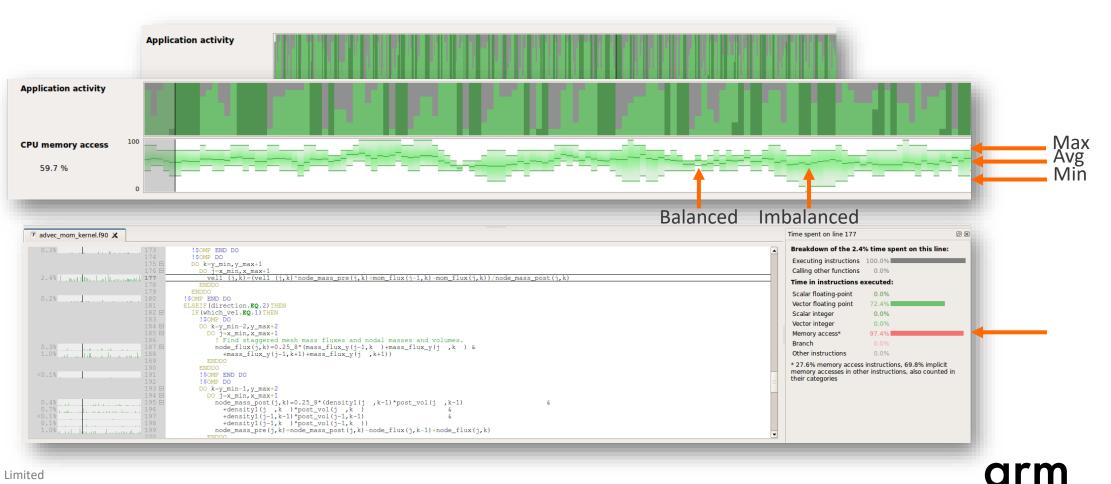
OpenMP imbalance: implicit synchronization





Understanding resource usage

• Memory accesses



Vectorization



© 2017 Arm Limited

Analyze the results

Running Performance Reports with CloverLeaf using 8 MPI tasks indicates that:

- Time spent in scalar ops is 14.7%
- Time spent in vector ops 18.9%

Summary: clover_leaf is Compute-bound in this configuration

Compute	93.4%	Time spent running application code. High values are usually good. This is very high ; check the CPU performance section for advice
MPI	6.6%	Time spent in MPI calls. High values are usually bad. This is very low ; this code may benefit from a higher process count
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is negligible ; there's no need to investigate I/O performance

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below.

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU 🔶		
A breakdown of th	e 93.4% CPU time:	
Scalar numeric ops	14.7%	
Vector numeric ops	18.9%	
Memory accesses	66.3%	

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

Little time is spent in vectorized instructions. Check the compiler's

vectorization advice to see why key loops could not be vectorized.

MPI

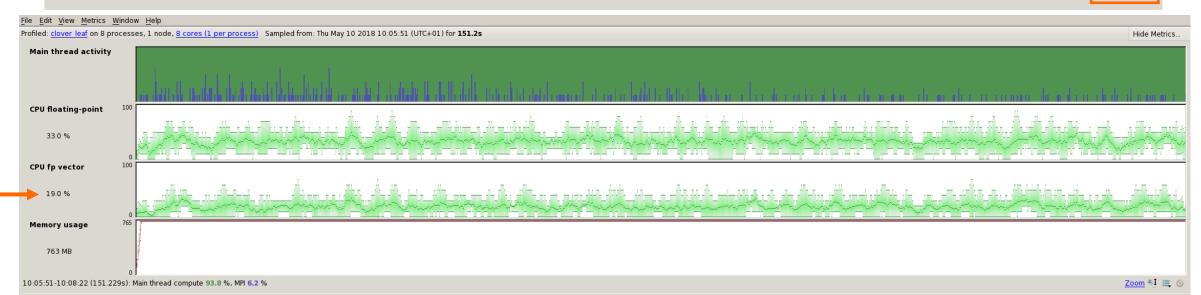
A breakdown of the 6.6% MPI tim	e:	
Time in collective calls	20.9%	
Time in point-to-point calls	79.1%	
Effective process collective rate	1.55 kB/s	
Effective process point-to-point rate	33.1 MB/s	

Most of the time is spent in point-to-point calls with a low transfer rate. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait.



How much of the code is vectorized?

Profiled: clover leaf on 8 processes, 1 node, 8 cores (1 per process) Sampled from: Thu May 10 2018 10:05:51 (UTC+01) for 151.2s



Main Thread Stacks			Main Thread Stacks				
Total core time	MPI	Function(s) on line	Source	Total core time	MPI	Function(s) on line	Source
		🗄 🌮 clover_leaf [program]	Jource			🖻 Þ clover_leaf [program] 🖻 💉 clover_leaf	PROGRAM clover_leaf
53.3% [[[]]]	2.0%	Ė ≠ clover_leaf ∉advection module∷advection	PROGRAM clover_leaf CALL advection()			i hydro advection_module∷advection advection_module::advection	CALL hydro CALL advection() . CALL advec_mom_driver(tile,xvel,direction,sweep_number)
16.5% <mark>(2016) 100 100 100 100 100 100 100 100 100 10</mark>	2.8% 0.5%	timestep_module∷timestep ⊕ pdv_module∷pdv	CALL timestep() CALL PdV(.TRUE.)			advec_mom_kernel_mod::advec	CALL advec_mom_ariver(tile,xvel,direction,sweep_number) CALL advec_mom_kernel(chunk%tiles(tile)%t_xmin, & CALL advec_mom_driver(tile,xvel,direction,sweep_number)
6.4% distantialitation kanalaktida kiinalitatiana 4.9% rahartiktiin distantiktianinga hisaalaine misiita	0.7%	⊕ pdv_module::pdv ⊕ accelerate module::accelerate	CALL PdV(.FALSE.) CALL accelerate()	9.8% kalmannaalinhiikkiinkeekkimkehike 9.1% romainaa kehinamatareiterkiin	i an	advec_cell_driver_module::advec advec_cell_driver_module::advec	CALL advec_cell_driver(tile,sweep_number,direction) CALL advec_cell_driver(tile,sweep_number,direction)
4.2% antestication internal contradication of the state o		⊕reset_field_module::reset_field ⊕flux calc module::flux calc	CALL reset_field()	7.1% addisectional autoritation tattibudiantilikantalaatas 6.6% aattaanaanaadadkamutakutakutakutakutakutakutakutakutakutak	alit.	advec_mom_driver_module::advec	 CALL advec_mom_driver(tile,yvel,direction,sweep_number) CALL advec_mom_driver(tile,yvel,direction,sweep_number) CALL update_halo (fields,2)
3.4% <u> </u>			CALL flux_calc()	0.9% 12.2% (which is the state of the	0.7% 111 3.2%	i ⊇ others i timestep_module∷timestep	CALL timestep()
				10.2% adducted uturitation and adduction 6.5% tableat duction adduction and a second adduction 4.8%	u 0.7%		CALL PdV(.TRUE.) CALL PdV(.FALSE.)
				4.8% uttractilder dürada, bedatilatildan det all det uttral det ut 4.2% en name te mond taktiletinder det	ada	accelerate_module::accelerate ⊕ reset_field_module::reset_field ⊕ flux_calc_module::flux_calc	CALL accelerate() CALL reset_field() CALL flux_calc()
				0.6%			

Showing data from 8,000 samples taken over 8 processes (1000 per process)

Where is the code vectorized?

) 🗙	🗉 advec_mom_driver.f90 🗵 🛛 🐨 advec_mom_kernel.f90 🔯 🖊	Time spent on line 159 & *
150	dif=donor	Breakdown of the 0.1% time spent on this line:
151	ELSE	
152	upwind=j-1	Executing instructions 100.0%
153	donor=j	Calling other functions 0.0%
154	downwind=j+1	5
155	dif=upwind	Time in instructions executed:
156	ENDIF	Scalar floating-point 63.6%
157	<pre>sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))</pre>	Vector floating point 0.0%
158	width=celldx(j)	
159	vdiffuw=vel1(donor,k)-vel1(upwind,k)	Scalar integer 18.2%
160 161	<pre>vdiffdw=vel1(downwind,k)-vel1(donor,k) limiter=0.0</pre>	Vector integer 0.0%
162 🖂	IF (vdiffuw*vdiffdw.GT.0.0) THEN	Memory access* 81.8%
163	auw=ABS (vdiffuw)	
164	adw=ABS(vdiffdw)	Branch 0.0%
165	wind=1.0 8	Other instructions 0.0%
166	IF (vdiffdw.LE.0.0) wind=-1.0_8	* 18.2% memory access instructions, 63.6% implicit memory accesses in other
167	limiter=wind*MIN (width*((2.0_8-sigma)*adw/widt	instructions, also counted in their categories
168	ENDIF	instated in their categories
169	advec_vel_s=vel1(donor,k)+(1.0-sigma)*limiter	
170	<pre>mom_flux(j,k)=advec_vel_s*node_flux(j,k)</pre>	
171	ENDDO	
172	ENDDO	
173	!\$OMP END DO	l
174	LSOMP DO	



Follow Performance Reports advice

advec_mom_kernel.f90

- •••
- 144 DO k=y_min,y_max+1
- 145 DO j=x_min-1,x_max+1 ←
- 146 IF(node_flux(j,k).LT.0.0)THEN
- 147 upwind=j+2
- 148 donor=j+1
- 149 downwind=j
- 150 dif=donor
- 151 ELSE
- 152 upwind=j-1
- 153 donor=j
- 154 downwind=j+1
- 155 dif=upwind
- 156 ENDIF
- 157 sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
- 158 width=celldx(j)
- 159 vdiffuw=vel1(donor,k)-vel1(upwind,k) ←
- 160 vdiffdw=vel1(downwind,k)-vel1(donor,k)

-fopt-info-vec-missed

advec_mom_kernel.f90:145: note: not vectorized: control flow in loop advec_mom_kernel.f90:145: note: bad inner-loop form. advec_mom_kernel.f90:145: note: not vectorized: Bad inner loop. advec_mom_kernel.f90:145: note: bad loop form. Analyzing loop at advec_mom_kernel.f90:145

advec_mom_kernel.f90:145: note: not vectorized: control flow in loop advec_mom_kernel.f90:145: note: bad loop form.



...

How well is the compiler vectorizing?

advec_mom_kernel.f90

- •••
- 144 DO k=y_min,y_max+1
- 145 DO j=x_min-1,x_max+1 ←
- 146 IF(node_flux(j,k).LT.0.0)THEN
- 147 upwind=j+2
- 148 donor=j+1
- 149 downwind=j
- 150 dif=donor
- 151 ELSE
- 152 upwind=j-1
- 153 donor=j
- 154 downwind=j+1
- 155 dif=upwind
- 156 ENDIF
- 157 sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
- 158 width=celldx(j)
- 159 vdiffuw=vel1(donor,k)-vel1(upwind,k)
- 160 vdiffdw=vel1(downwind,k)-vel1(donor,k)

-qopt-report=2

LOOP BEGIN at advec_mom_kernel.f90(145,9) <Peeled loop for vectorization> remark #25456: Number of Array Refs Scalar Replaced In Loop: 2 LOOP END

LOOP BEGIN at advec_mom_kernel.f90(145,9) remark #15300: LOOP WAS VECTORIZED LOOP END

LOOP BEGIN at advec_mom_kernel.f90(145,9) <Remainder loop for vectorization> LOOP END



...

Analyze the results

Running Performance Reports with CloverLeaf using 8 MPI tasks indicates that:

- Time spent in scalar ops is 4.8%
- Time spent in vector ops 28.2%

Summary: clover_leaf is Compute-bound in this configuration

Compute	92.9%	Time spent running application code. High values are usually good. This is very high ; check the CPU performance section for advice
MPI	7.1%	Time spent in MPI calls. High values are usually bad. This is very low ; this code may benefit from a higher process count
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is negligible ; there's no need to investigate I/O performance

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below.

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of th	ne 92.99	% CPU time:
Scalar numeric ops	4.8%	I
Vector numeric ops	28.2%	
Memory accesses	67.0%	

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 7.1% MPI time	e:	
Time in collective calls	24.4%	
Time in point-to-point calls	75.6%	
Effective process collective rate	1.35 kB/s	
Effective process point-to-point rate	33.9 MB/s	

Most of the time is spent in point-to-point calls with a low transfer rate. This can be caused by inefficient message sizes, such as many small messages, or by imbalanced workloads causing processes to wait.



Where is the code vectorized?

dono. down dif= ENDIF sigma= width= vdiffud limite IF(vdi auw=2 adw=2 wind IF(vdi IF(vdi auw=2 adw=2 wind IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(<pre>nd=j-1 r=j wind=j+1 upwind ABS(node_flux(j,k))/(node_mass_pre(donor,) ccelldx(j) w=vel1(donor,k)-vel1(upwind,k) w=vel1(donor,k)</pre>	k))		Time spent on line 159 Breakdown of the Executing instructions Calling other functions		line:
dono. down down dif= ENDIF sigma= width= vdiffut limite IF(vdi auw=] adw=j wind IF(vdi auw=] adw=j wind IF(vdi ENDIF advec_ mom_fl ENDIF advec_ mom_fl ENDDO ENDDO DO k=y_min DO j=x_m; vel1(c) ENDDO	<pre>r=j⁻ wind=j+1 upwind ABS(node_flux(j,k))/(node_mass_pre(donor,) celldx(j) w=vel1(donor,k)-vel1(upwind,k) w=vel1(downwind,k)-vel1(donor,k) r=0.0 ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffuw) ABS(vdiffuw) aBS(vdiffuw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/widd))</pre>	k))		Executing instructions	100.0%	line:
down dif= ENDIF sigma= width= vdiffu vdiffu limite □ IF(vdi auw=J adw=J wind IF(vdi IF(vdi auw=J adw=J wind IF(vol	<pre>wind=j+1 upwind ABS(node_flux(j,k))/(node_mass_pre(donor,) ccelldx(j) w=vel1(donor,k)-vel1(upwind,k) w=vel1(downwind,k)-vel1(donor,k) er=0.0 ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffdw) als(vdiffdw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/widd)) </pre>	k))		Executing instructions	100.0%	line:
dif= ENDIF sigma= width= vdiffd limite IF(vdi auw=J adw=J wind: IF(v, limite IF(vdi auw=J adw=J wind: IF(v, limite IF(vdi ENDIF advec_ mom_fl ENDDO ENDDO SomP DO DO k=y_min DO j=x_m: vel1 (j ENDDO	<pre>upwind ABS(node_flux(j,k))/(node_mass_pre(donor,) ccelldx(j) w=vel1(donor,k)-vel1(upwind,k) w=vel1(downwind,k)-vel1(donor,k) rr=0.0 ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffdw) aBS(vdiffdw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/widd))</pre>	k))		-		
sigma= width= vdiffu vdiffu limite IF(vdi auw=J adw=J wind IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF(vdi) IF	<pre>scelldx(j) w=vel1(donor,k)-vel1(upwind,k) w=vel1(downwind,k)-vel1(donor,k) er=0.0 ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffdw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/widd))</pre>	k))		-		
width= vdiffu vdiffd limite IF(vdi auw=J adw=j wind: IF(vv limi ENDIF advec_ mom_fl ENDDO ENDDO Source Sector Source Sector	<pre>scelldx(j) w=vel1(donor,k)-vel1(upwind,k) w=vel1(downwind,k)-vel1(donor,k) er=0.0 ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffdw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/widd))</pre>	k))		-		
vdiffut vdiffut limite limite IF(vdi auw=1 adw=i wind: IF(vul limi ENDIF advec_1 mom_fl ENDDO ENDDO I\$OMP DO DO k=y_min D0 j=x_m: vel1 (j ENDDO	<pre>w=vel1(donor,k)-vel1(upwind,k) w=vel1(downwind,k)-vel1(donor,k) rr=0.0 ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffuw) ABS(vdiffuw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/wid)</pre>			Calling other functions	0.0%	
<pre>limite If(vdi auw=l adw=l adw=l adw=l adw=l limi ENDIF advec_l mom_fl ENDD0 ENDD0 I\$0MP END I\$0MP END D0 l\$0 k=y_min D0 j=x_m: vel1 (j ENDD0</pre>	er=0.0 ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffdw) aBS(vdiffdw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/wid		I	canny outer rancaons		
□ IF (vdi a uw=1 a dw=1 wind: IF (vdi umi ENDIF advec_1 mom_fl ENDDO ENDDO ENDDO Soft ENDDO DO k=y_min DO j=x_m; vel1 (j ENDDO	<pre>ffuw*vdiffdw.GT.0.0)THEN ABS(vdiffuw) ABS(vdiffuw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/widd)</pre>		I			
auw=2 adw=2 wind IF(vd limi ENDIF advec_ mom_fl ENDDO ENDDO ENDDO I\$OMP_END I\$OMP_END I\$OMP_DO DO k=y_min DO j=x_m: vel1() ENDDO	ABS(vdiffuw) ABS(vdiffdw) =1.0_8 diffdw.LE.0.0) wind=-1.0_8 ter=wind*MIN(width*((2.0_8-sigma)*adw/wid			Time in instantion		
wind: IF(vv. limi ENDIF advec_ mom_fl ENDDO ENDDO i\$OMP END i\$OMP END bo k=y_min Do j=x_m: vell (j ENDDO	=1.0_8 diffdw. LE .0.0) wind=-1.0_8 ter=wind* MIN (width*((2.0_8-sigma)*adw/wid			Time in instruction	is executed:	
IF (v4 limi ENDIF advec_ mom_fl ENDDO ENDDO I\$OMP END I\$OMP END I\$OMP DO DO k=y_min DO j=x_m: vel1 (j ENDDO	diffdw. LE .0.0) wind=-1.0_8 ter=wind* MIN (width*((2.0_8-sigma)*adw/wid					
limi ENDIF advec_ mom_fl ENDDO ENDDO !\$OMP END !\$OMP END !\$OMP PO DO k=y_min D 0 j=x_m: vel1 (j ENDDO	ter=wind* MIN (width*((2.0_8-sigma)*adw/wid			Scalar floating-point	0.0%	
advec_ mom_fl ENDDO ENDDO I\$OMP END I\$OMP DO DO k=y_min DO j=x_m: vell (; ENDDO	vel s=vel1(dopor.k)+(1.0-sigma)*limiter	dth+(1.0_8+sigma)*auw/ce	elldx(dif))/6.0_8,auw,adw)	5.		
mom_fl ENDDO ENDDO !\$OMP END !\$OMP DO □ DO k=y_min □ DO j=x_m: vell (; ENDDO				Vector floating point	28.6%	
ENDDO \$OMP END : \$OMP DO DO k=y_min DO j=x_m: vel1 (ENDDO	<pre>ux(j,k)=advec_vel_s*node_flux(j,k)</pre>			rector notating point		
!\$OMP END !\$OMP DO □ DO k=y_min □ DO j=x_m: vel1 () ENDDO				Scalar integer	0.0%	
<pre>!\$OMP DO DO k=y_min DO j=x_m: vel1 () ENDDO</pre>	DO			bealar meeger	0.070	
DO j=x_m: vel1 (j ENDDO				Vector integer	0.0%	
vel1 (ENDDO				vector integer	0.070	
ENDDO	j,k)=(vel1 (j,k)*node_mass_pre(j,k)+mom_fl	lux(j-1,k)-mom flux(j,k)))/node mass post(j,k)	Memory access	68.6%	
ENDDO				Memory access	00.070	
\$0MP END DO				Branch		
ELSEIF(dired	ction. EQ.2) THEN			Dranch		
IF (which_v \$0MP DO	vel.EQ.1) THEN			Other instructions	2.9%	
	in-2,y_max+2		Main Thread Stacks	Other instructions	2.970	
DO j=x_	_min,x_max+1		Total core time ∇ MPI			
Thread Stacks						
core time	∇ MPI Function(s) on line	Source				
	🖻 🥵 clover_leaf [program]			advec_mom_driver_module::advec CALL adve	c_mom_driver(tile,xvel,direction,sweep_number)	
	🖻 💉 clover_leaf	PROGRAM clover_leaf	9.7% namanalandintina induntintindin halada 8.9% <u>uun</u> tatatat	advec_mom_driver_module::advec CALL advec advec cell driver module::advec ce CALL adve	c_mom_driver(tile,xvel,direction,sweep_number) c cell driver(tile,sweep number,direction)	
.6% ^{[[]}	e hydro	CALL hydro	8.5% and an diality of a state of a state of the state of	advec_cell_driver_module::advec_ce CALL advec	c_cell_driver(tile,sweep_number,direction)	
		CALL advection() CALL timestep()	6.8% aattalmatina katalmi katalmi katalmi katalmi katalmi 6.2% caatan katalmi katalmi katalmi katal katali		c_mom_driver(tile,yvel,direction,sweep_number) c_mom_driver(tile,yvel,direction,sweep_number)	
.2% addam lathanta in in an abhain in a		CALL PdV(.TRUE.)	1.5% allotable allålatend tille at at han an 1.4%	update_halo_module::update_halo CALL update_halo	te_halo(fields,2)	
.5%		CALL PdV(.FALSE.)	0.8% 0.6% 0.6% 0.6% 0.6%		step()	
.8%		CALL accelerate()	10.2% itimatang aktivan halanda an halan bilan b	pdv_module::pdv CALL PdV	.TRUE.)	
.2% 🖬 ռուս և ստանեն մասնին Ասև			6.4% utocculations.scools.cd.abundutationing.abundutationing.abundutation 4.9% calaatatiana distantiistaannaaliis.aduuna matuu			
.2%		CALL flux_calc()	4.2% autobalantation and a state and the state of the sta	reset_field_module::reset_field CALL reset	t_field()	1
.6%			3.4% Itaniatadanialitadianaa addadaa addadaa 1.2%	flux_calc_module::flux_calc CALL flux	calc()	
ving data from 8,000 san	nples taken over 8 processes (1000 per process)					

Thank You! Danke! Merci! 谢谢! ありがとう! **Gracias!** Kiitos!

