

Profiling sequential programs



Key question

- How do I know where my code spends most of its time?
 - This is called “profiling”
 - Many (free and commercial) tools exist
- Baseline tool: GNU gprof
 - Supported by GCC and Intel compilers (and others)
- Linux “perf” infrastructure
- Based on the profile, optimization can be planned
 - Reduction of work
 - Doing work more efficiently

Profiling with gprof

Caveat: all-new profiling tool “gprofng” upcoming!



Profiling with gprof

- Basic sequential profiling tool under Linux: **gprof**
- Compiling for a profiling run

```
$ icx -pg .....
```

- After running the binary, a file **gmon.out** is written to the current directory
- Human-readable output:

```
$ gprof a.out
```

- Inlining should be disabled for profiling
 - But then the executed code isn't what it should be...

Profiling with gprof: Example

- Example with wrapped `double` class:

```
class D {  
    double d;  
public:  
    D(double _d=0) : d(_d) {}  
    D operator+(const D& o) {  
        D r;  
        r.d = d+o.d;  
        return r;  
    }  
    operator double() {  
        return d;  
    }  
};
```

Main program:

```
const int n=10000000;  
D a[n],b[n];  
D sum;  
  
for(int i=0; i<n; ++i)  
    a[i] = b[i] = 1.5;  
  
double s = timestamp();  
for(int k=0; k<10; ++k) {  
    for(int i=0; i<n; ++i)  
        sum = sum + a[i] + b[i];  
}
```

Profiling with gprof: Example profiler output

- `icpx -O3 -pg perf.cc`

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
101.01	0.41	0.41				main

- `icpx -O3 -fno-inline -pg perf.cc`

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
46.44	0.59	0.59	200000000	2.93	4.48	D::operator+(D const&)
29.63	0.96	0.37	240000001	1.56	1.56	D::D(double)
24.82	1.27	0.31				main

- But where did the time *actually* go?
 - Butterfly (callgraph) profile also available
 - Real problem also with use of libraries (STL!)
 - Sometimes you have to roll your own little profiler (see later!)

Flat profile

Called how often?

Time not including callees

Time including callees

Each sample counts as 0.000976562 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
66.43	7.96	7.96	1	7.96	9.64	hamilt_
11.15	9.30	1.34	1	1.34	1.34	outwin_
8.14	10.28	0.98	1	0.98	1.07	hns_
2.86	10.62	0.34	5266813	0.00	0.00	zheevx2_
1.20	10.76	0.14				__libm_error_support
1.04	10.89	0.12				zher2m_
0.90	11.00	0.11				cvtas_s_to_a
0.75	11.09	0.09				select_
0.38	11.13	0.05				cvt_ieee_t_to_text_ex
0.37	11.18	0.04	445351	0.00	0.00	seclit_

Butterfly (call graph) profile

granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

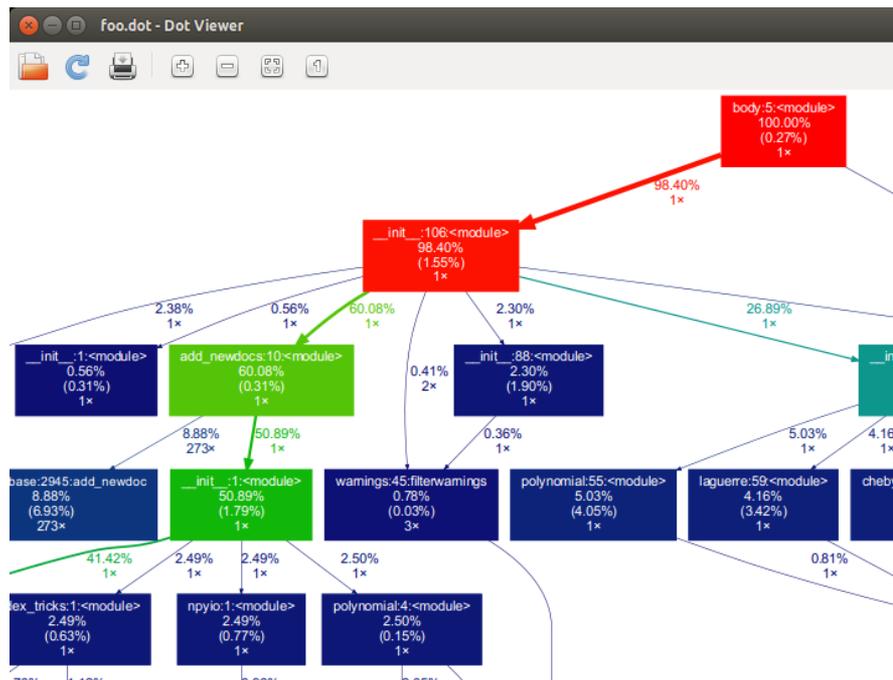
index	% time	self	children	called	name
[1]	100.0	0.00	0.05		<spontaneous>
		0.00	0.05	1/1	start [1]
		0.00	0.00	1/2	main [2]
		0.00	0.00	1/1	on_exit [28]
		0.00	0.00	1/1	exit [59]

[2]	100.0	0.00	0.05	1/1	start [1]
		0.00	0.05	1	main [2]
		0.00	0.05	1/1	report [3]

[3]	100.0	0.00	0.05	1/1	main [2]
		0.00	0.03	8/8	report [3]
		0.00	0.01	1/1	timelocal [6]
		0.00	0.01	1/1	print [9]
		0.00	0.01	9/9	fgets [12]
		0.00	0.00	12/34	strncmp <cycle 1> [40]
		0.00	0.00	8/8	lookup [20]
		0.00	0.00	1/1	fopen [21]
		0.00	0.00	8/8	chewtime [24]
		0.00	0.00	8/16	skipspace [44]

Visualizing the butterfly profile

- **Gprof2Dot** converts gprof output to graphviz “dot” file
 - <https://github.com/jrfonseca/gprof2dot>
- View dot file with, e.g., **xdot**



Profiling MPI programs with gprof

- By design, gprof is a tool for **serial code**
 - It can, however, be convinced to write a trace file that contains the PID in its name

```
$ GMON_OUT_PREFIX=foo mpirun -np 5 ./a.out
[...]  
$ ls  
a.out  foo.28219  foo.28220  foo.28221  foo.28222  foo.28223  
$ gprof a.out foo.28219
```

- **Accumulating** individual files:

```
$ gprof --sum a.out foo.* # generates gmon.sum  
$ gprof a.out gmon.sum
```

- Take care – all values are summed up across processes!

Sampling-based profiling with Linux perf



Sampling

- **Interrupts** program in regular intervals
- Records **data**
 - **Where** is the program right now (address of “current” instruction)?
 - program function
 - line of code
 - machine instruction
 - What does the **call stack** look like?
 - How did execution get “here”?
- **Results**
 - Histogram of “**how much time spent where**”
 - How much time is spent **along a particular call path**

Advantages vs. gprof:

- Works on any binary without recompile
- Also captures OS and runtime symbols
- Also works with multi-threaded code

Simple runtime profile with Linux perf

Compile with (Intel compiler; options also work for GCC):

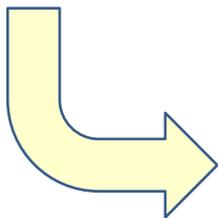
```
$ icx -g -fno-inline -fno-omit-frame-pointer ...
```

Call executable wrapped with perf, generated file `perf.data`:

```
$ perf record -g ./a.out
```

Analyze the results with:

```
$ perf report
```



```
Samples: 113K of event 'cycles:Pu', Event count (approx.): 63028635012
```

	Children	Self	Command	Shared Object	Symbol
+	100,00%	0,00%	a.out	libc.so.6	[.] __libc_start_call_main
+	100,00%	0,00%	a.out	a.out	[.] main
+	96,01%	0,00%	a.out	a.out	[.] CG
+	57,51%	57,51%	a.out	a.out	[.] axpby
+	22,14%	22,14%	a.out	a.out	[.] dotProduct
+	19,06%	19,06%	a.out	a.out	[.] applyStencil
+	2,14%	0,00%	a.out	a.out	[.] computeResidual
+	0,69%	0,64%	a.out	a.out	[.] __svml_sin4_19
	0,46%	0,41%	a.out	a.out	[.] init
	0,23%	0,23%	a.out	a.out	[.] __intel_avx_rep_memcpy
	0,02%	0,00%	a.out	[unknown]	[.] 0xfffffffffb2800c0

Output of `perf report`

Inclusive time for function

Self time for function

Event used for profiling

```
Samples: 113K of event 'cycles:Pu', Event count (approx.): 63028635012
```

Children	Self	Command	Shared Object	Symbol
+ 100,00%	0,00%	a.out	libc.so.6	[.] __libc_start_call_main
+ 100,00%	0,00%	a.out	a.out	[.] main
+ 96,01%	0,00%	a.out	a.out	[.] CG
+ 57,51%	57,51%	a.out	a.out	[.] axpby
+ 22,14%	22,14%	a.out	a.out	[.] dotProduct
+ 19,06%	19,06%	a.out	a.out	[.] applyStencil
+ 2,14%	0,00%	a.out	a.out	[.] computeResidual
+ 0,69%	0,64%	a.out	a.out	[.] __svml_sin4_l9
0,46%	0,41%	a.out	a.out	[.] init
0,23%	0,23%	a.out	a.out	[.] __intel_avx_rep_memcpy
0,02%	0,00%	a.out	[unknown]	[.] 0xfffffffffffb2800c00

“profile browser” allows drilling into call tree if “-g” option was given for `perf record` (hit “h” for help)

Options for reporting with perf

- Show perf.data in an ncurses browser (TUI) if possible:
`$ perf report`
- Show perf.data with a column for sample count:
`$ perf report -n`
- Show perf.data as a text report, with data coalesced and percentages:
`$ perf report --stdio`
- Disassemble and annotate instructions with percentages (needs some debuginfo):
`$ perf annotate --stdio`

Some general remarks about profiling



Manual profiling with a timer function

- Measuring walltime on UNIX (-like) systems
 - Stay away from CPU time – it's evil!
 - Use `clock_gettime()` to obtain wall-clock time stamp:

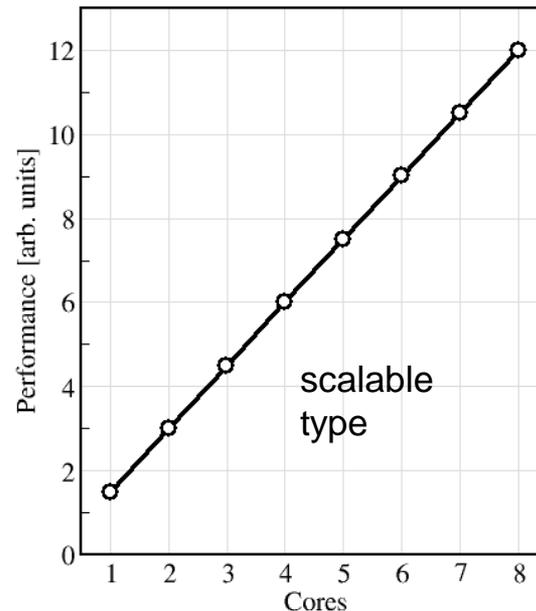
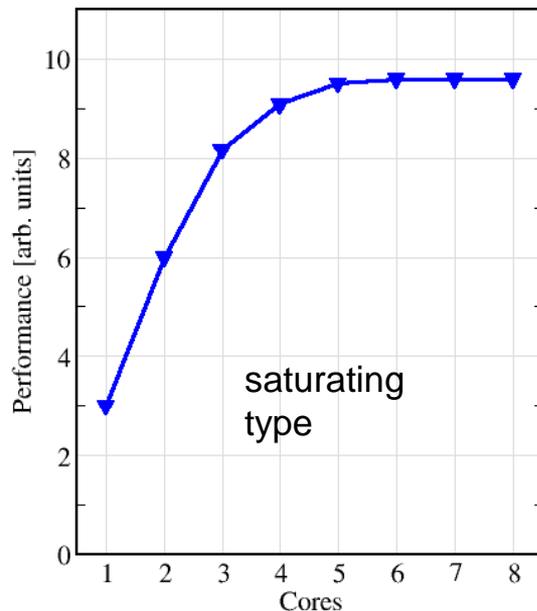
```
#include <time.h>

double getTimeStamp()
{
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9;
}

double gettimestamp_()
{
    return getTimeStamp();
}
```

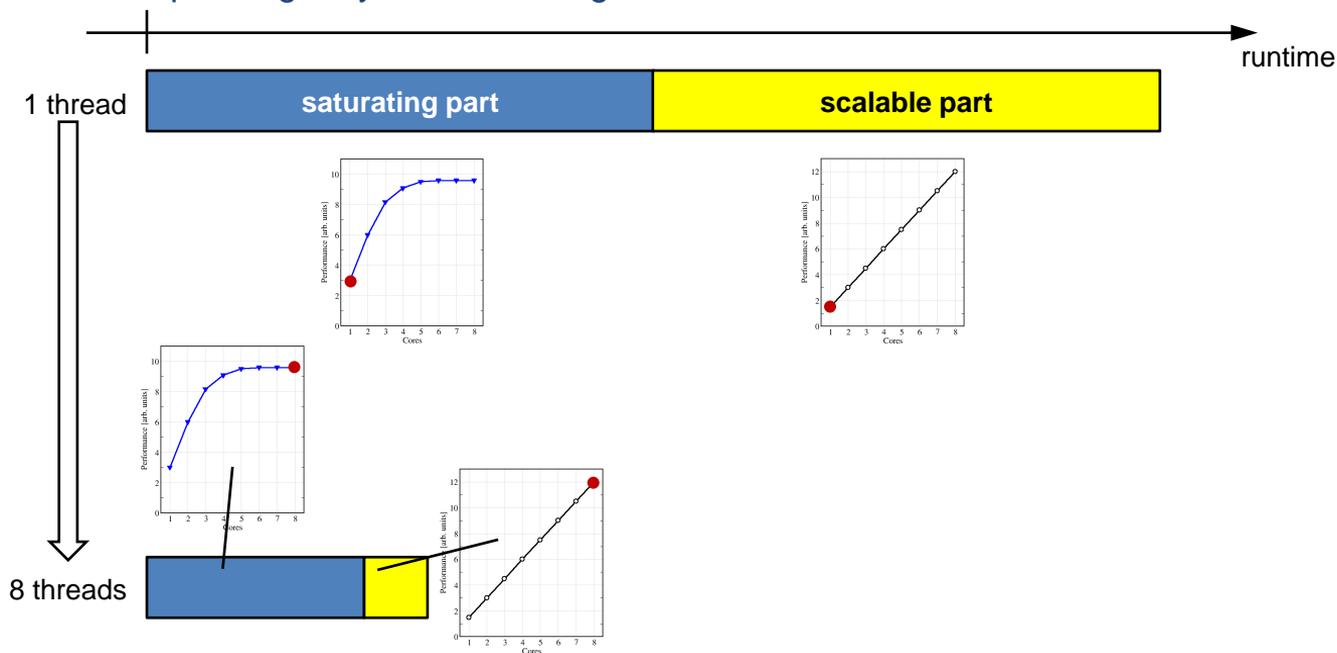
Consequences from the saturation pattern for profiling

Clearly distinguish between “**saturation**” and “**scalable**” performance on the chip level



Consequences from the saturation pattern for profiling

- Some bottlenecks only show up in parallel execution!
- Code profile for single thread \neq code profile for multiple threads
 - \rightarrow Single-threaded profiling may be misleading



MPI program tracing with Intel Trace Analyzer/Collector (ITAC)

Georg Hager, Alireza Ghasemi, Ayesha Afzal

Intel Trace Analyzer and Collector

Event-based tool recording user function calls and MPI communication calls

GUI for advanced visualization

Summary: xhpcg.stf
Total time: 1.88e+04 sec. Resources: 9 processes, 1 node.

Ratio
This section represents a ratio of all MPI calls to the rest of your code in the application.

- Serial Code - 1.88e+04 sec 99.1 %
- MPI calls - 160 sec 0.8 %

Top MPI functions
This section lists the most active MPI functions from all MPI calls in the application.

MPI Function	Duration
MPI_Allreduce	75.8 sec
MPI_Wait	75.8 sec
MPI_Send	7.83 sec
MPI_Recv	0.683 sec
MPI_Wtime	0.14 sec

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

to optimize node-level performance use:

- Intel® VTune™ Amplifier XE** for:
 - algorithmic level tuning with hotspots and threading efficiency analysis;
 - microarchitecture level tuning with general exploration and bandwidth analysis;
- Intel® Advisor** for:
 - vectorization optimization and thread prototyping.

For more information, see documentation for the respective tool:
[Analyzing MPI applications with Intel® VTune™ Amplifier XE](#)
[Analyzing MPI applications with Intel® Advisor](#)

Show Summary Page when opening a tracefile

Flat Profile

Name	Duration (%)	Duration
Wait at Barrier	0.15%	100.472e-3
Late Sender	0.00%	29.609e-3
Late Receiver	0.00%	1.45e-3

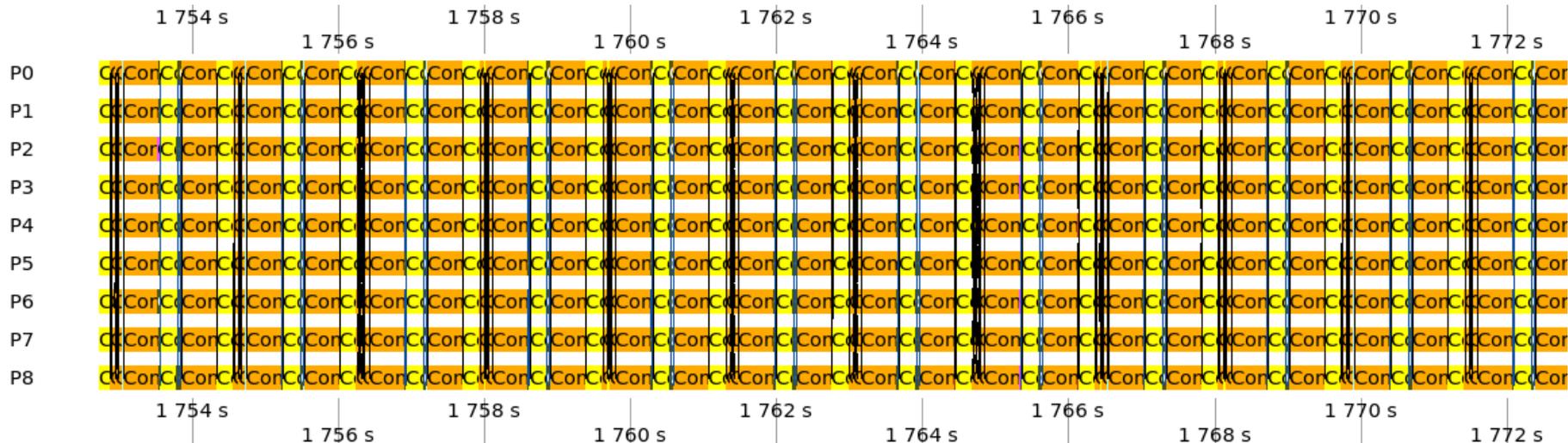
Performance Issue: Wait at Barrier

Description: Affected Processes: P1

Total Time [s] (Sender by Receiver)

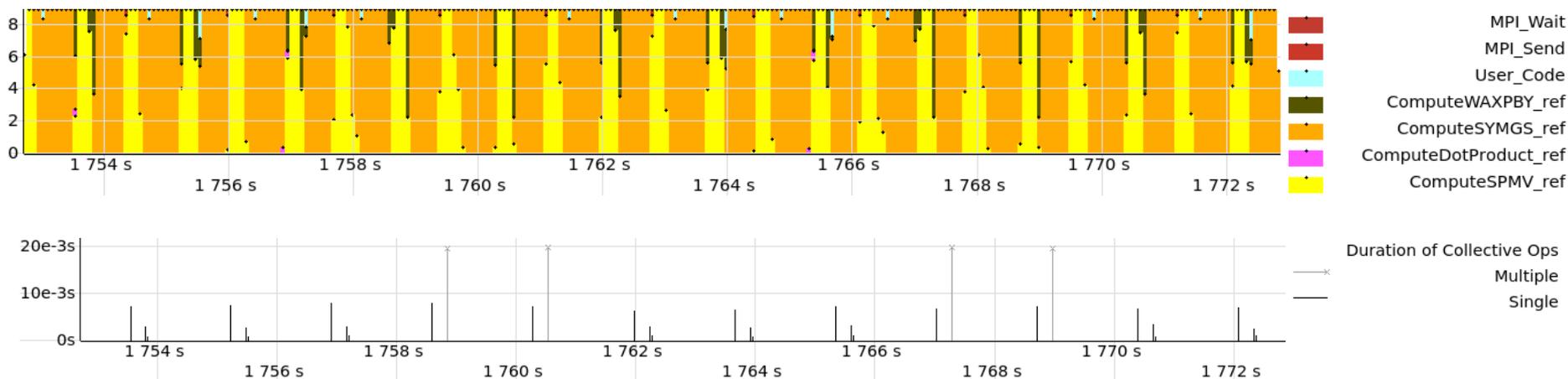
	P0	P1	P2	P3	P4
MPI_Allreduce	18.43e-3	2.223e-3	43.023e-3	4.8e-3	4.8e-3
Sum	18.43e-3	2.223e-3	43.023e-3	4.8e-3	4.8e-3
Mean	18.43e-3	2.223e-3	43.023e-3	4.8e-3	4.8e-3
StdDev	0	0	0	0	0

Event timeline view



- Timeline of MPI and user function execution
- Message visualization
- Context menu provides details on functions/messages
- Zoom/pan

Quantitative and qualitative timelines



- Time spent in different MPI/user functions across processes
- Duration of certain things (collectives, PtP)

Performance advice

Performance Issue	Duration (%)	Duration
Wait at Barrier	0.00%	29.882e-3 s
Late Broadcast	0.00%	2e-6 s

[Show advanced..](#)

Description	Affected Processes	Source Locations (Root Causes)
Wait at Barrier		

This problem occurs when barrier collective operations (such as MPI_Barrier or all-to-all operations such as MPI_Alltoall which also result in barrier) are not synchronized. The result is that these operations wait for the last barrier operation to start. Usually this problem indicates load imbalance in a program.

To resolve this problem, make sure that barrier operations are called at approximately the same time by all processes. This can be done by adding computation prior to the earlier barrier calls or by lessening the computation prior to the later barrier calls.

Affected Processes shows the distribution of the issue duration per process.

Source Locations shows source locations (with corresponding durations) that are root causes of this problem.

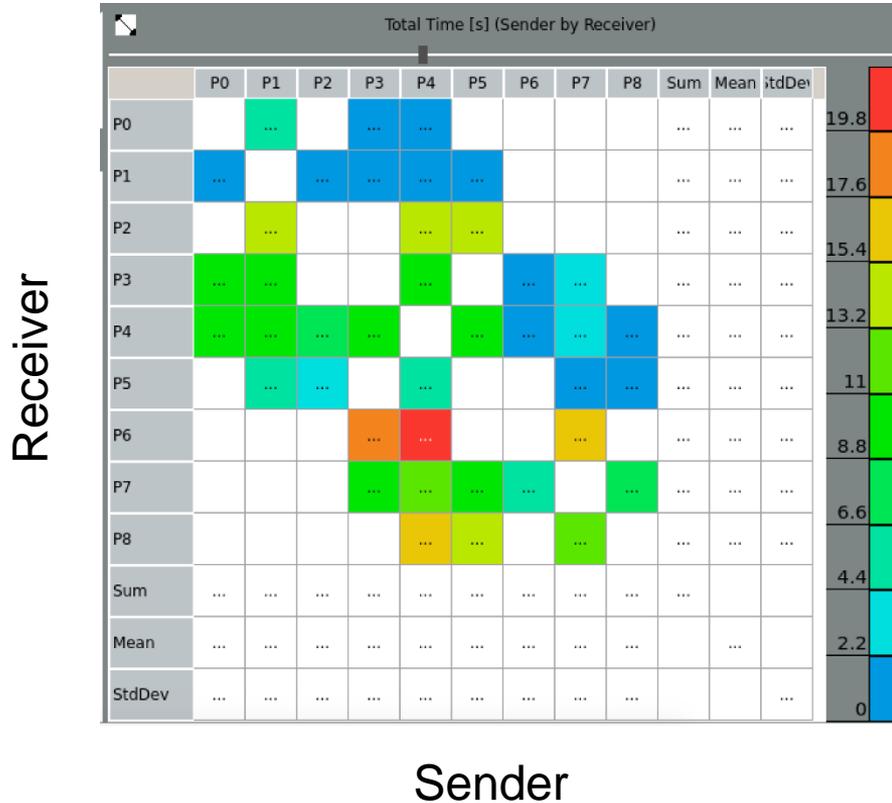
Performance Issue	Duration (%)	Duration
Wait at Barrier	0.00%	29.882e-3 s
Late Broadcast	0.00%	2e-6 s

[Show advanced..](#)

Description	Affected Processes	Source Locations (Root Causes)
Process	Duration (%)	Duration
Process 6	0.00%	6.287e-3 s
Process 2	0.00%	5.742e-3 s
Process 4	0.00%	4.4e-3 s
Process 8	0.00%	4.057e-3 s
Process 3	0.00%	3.992e-3 s
Process 0	0.00%	1.988e-3 s
Process 7	0.00%	1.897e-3 s
Process 5	0.00%	1.519e-3 s

Context-sensitive advice on typical performance patterns

Message profile



- Who sends how much to whom?
- How long does it take?
- Effective bandwidth?

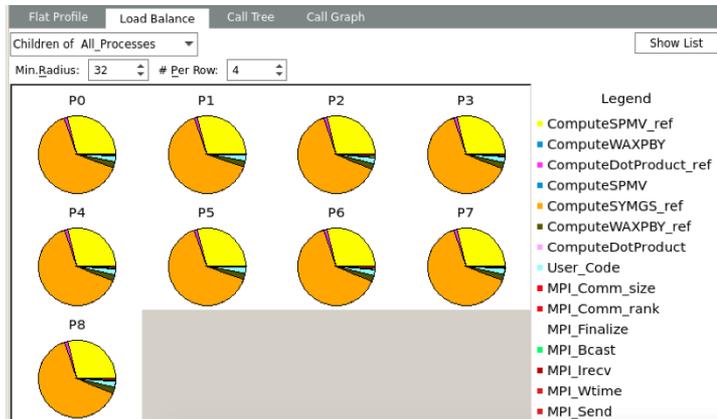
Collective operations profile

Total Time [s] (Collective Operation by Process)													
	P0	P1	P2	P3	P4	P5	P6	P7	P8	Sum	Mean	StdDev	
MPI_Bcast	5e-6	7e-6	7e-6	7e-6	7e-6	7e-6	6e-6	7e-6	7e-6	60e-6	6.66667e-6	666.667e-9	13.5
MPI_Allreduce	6.98827	2.41008	14.1332	9.46671	9.80818	2.28141	12.1689	7.89127	10.6684	75.8164	8.42405	3.81376	12
Sum	6.98828	2.41009	14.1332	9.46671	9.80818	2.28142	12.1689	7.89127	10.6684	75.8165			10.5
Mean	3.49414	1.20504	7.06659	4.73336	4.90409	1.14071	6.08444	3.94564	5.33422		4.21203		9
StdDev	3.49413	1.20504	7.06658	4.73335	4.90409	1.1407	6.08444	3.94563	5.33422			5.00135	7.5
													6
													4.5
													3
													1.5
													0

- Time spent in collective call
- Data volume sent/received

Functions profile, call tree/graph, load imbalance

Name	TSelf	TSelf	TTotal	#Calls	TSelf/Call
All_Processes					
All_Processes					
ComputeSPMV_ref	5.46742e+3 s		5.51519e+3 s	44244	123.574e-3 s
ComputeWAXPBY	53.862e-3 s		475.234 s	31995	1.68345e-6 s
ComputeDotProduct_ref	264.986 s		329.014 s	33408	7.93181e-3 s
ComputeSPMV	15.826e-3 s		2.49945e+3 s	10863	1.45687e-6 s
ComputeSYMGS_ref	11.9892e+3 s		12.0259e+3 s	76608	156.5e-3 s
ComputeWAXPBY_ref	495.263 s		495.263 s	33345	14.8527e-3 s
ComputeDotProduct	31.118e-3 s		315.682 s	32049	970.951e-9 s
User_Code	432.994 s		18.8104e+3 s	9	48.1104 s
MPI_Comm_size	132.739e-3 s		132.739e-3 s	120861	1.09828e-6 s
MPI_Comm_rank	63.288e-3 s		63.288e-3 s	120861	523.643e-9 s
MPI_Finalize	4.043e-3 s		4.043e-3 s	9	449.222e-6 s
MPI_Bcast	60e-6 s		60e-6 s	9	6.66667e-6 s
MPI_Recv	682.73e-3 s		682.73e-3 s	537120	1.27109e-6 s
MPI_Wtime	140.191e-3 s		140.191e-3 s	223353	627.666e-9 s
MPI_Send	7.83291 s		7.83291 s	14.5832e-6 s	
MPI_Allreduce	75.8164 s		75.8164 s	33534	2.26088e-3 s
MPI_Wait	75.7904 s		75.7904 s	537120	141.105e-6 s



Name	TSelf	TSelf	TTotal	#Calls	TSelf/Call
All_Processes					
User_Code	432.994 s		18.8104e+3 s	9	48.1104 s
MPI_Bcast	60e-6 s		60e-6 s	9	6.66667e-6 s
MPI_Comm_rank	1e-6 s		1e-6 s	9	111.111e-9 s
MPI_Comm_size	3e-6 s		3e-6 s	9	333.333e-9 s
MPI_Wtime	95.84e-3 s		95.84e-3 s	156537	612.251e-9 s
MPI_Allreduce	11.8325 s		11.8325 s	126	93.909e-3 s
ComputeSPMV_ref	2.97267e+3 s		3.01576e+3 s	33381	89.0527e-3 s
MPI_Comm_size	36.969e-3 s		36.969e-3 s	33381	1.10749e-6 s
MPI_Comm_rank	17.688e-3 s		17.688e-3 s	33381	529.882e-9 s
MPI_Recv	175.253e-3 s		175.253e-3 s	148360	1.18127e-6 s
MPI_Send	2.13018 s		2.13018 s	148360	14.3582e-6 s
MPI_Wait	40.7319 s		40.7319 s	148360	274.548e-6 s
ComputeSYMGS_ref	11.9892e+3 s		12.0259e+3 s	76608	156.5e-3 s
MPI_Comm_size	82.736e-3 s		82.736e-3 s	76608	1.07999e-6 s
MPI_Comm_rank	38.209e-3 s		38.209e-3 s	76608	498.76e-9 s
MPI_Recv	420.928e-3 s		420.928e-3 s	340480	1.23628e-6 s
MPI_Send	4.18757 s		4.18757 s	340480	12.299e-6 s
MPI_Wait	31.9995 s		31.9995 s	340480	93.9836e-6 s
ComputeWAXPBY_ref	20.0829 s		20.0829 s	1350	14.8762e-3 s
ComputeDotProduct_ref	10.7614 s		13.363e s	1359	7.91865e-3 s
MPI_Wtime	1.935e-3 s		1.935e-3 s	2718	711.921e-9 s
MPI_Allreduce	2.60027 s		2.60027 s	1359	1.91337e-3 s
ComputeSPMV	15.826e-3 s		2.49945e+3 s	10863	1.45687e-6 s
ComputeSPMV_ref	2.49475e+3 s		2.49943e+3 s	10863	229.656e-3 s
MPI_Comm_size	13.031e-3 s		13.031e-3 s	10863	1.19958e-6 s
MPI_Comm_rank	7.39e-3 s		7.39e-3 s	10863	680.291e-9 s
MPI_Recv	86.549e-3 s		86.549e-3 s	48280	1.79265e-6 s
MPI_Send	1.51516 s		1.51516 s	48280	31.388e-6 s
MPI_Wait	3.059 s		3.059 s	48280	63.359e-6 s
ComputeWAXPBY	53.862e-3 s		475.234 s	31995	1.68345e-6 s

Name	TSelf	TSelf	TTotal	#Calls	TSelf/Call
All_Processes					
Callers					
ComputeSPMV_ref called by ComputeSPMV	2.49475e+3 s		2.49943e+3 s	10863	229.656e-3 s
ComputeSPMV_ref called by User_Code	2.97267e+3 s		3.01576e+3 s	33381	89.0527e-3 s
ComputeSPMV_ref	5.46742e+3 s		5.51519e+3 s	44244	123.574e-3 s
Callees					
ComputeSPMV_ref calling MPI_Comm_size					
ComputeSPMV_ref calling MPI_Comm_rank					
ComputeSPMV_ref calling MPI_Recv					
ComputeSPMV_ref calling MPI_Send					

Options for taking traces

- **Caveat: Tracing can generate vast amounts of data!**
- Compiler switches (only works with legacy Intel compiler and wrappers [mpiicc, mpiicpc, mpiifort])
 - **-trace** # record MPI calls (also possible with mpirun/mpiexec)
 - **-tcollect -trace** # record MPI and user code function calls
potential of large overhead and large trace size
 - **-tcollect-filter=func.txt -tcollect -trace** # filter file

func.txt example

```
'.*' OFF  
'.*ComputeDotProduct.*' ON  
'.*ComputeSYMGS.*' ON  
'.*ComputeSPMV.*' ON  
'.*ComputeWAXPBY.*' ON
```

More (important) configuration options

Environment variable	Default	Description
VT_FLUSH_PREFIX	... depends	control directly for temporary flush files
VT_LOGFILE_PREFIX	current working directory	control directly for physical trace information files
VT_LOGFILE_FORMAT	STF	SINGLESTF: rolls all trace files into one file (.single.stf)
VT_LOGFILE_NAME	\${binary}.stf	control the name for the trace file
VT_MEM_BLOCKSIZE	64 KB	trace data in chunks of main memory
VT_MEM_FLUSHBLOCKS	1024	flushing is started when the number of blocks in memory exceeds this threshold
VT_MEM_MAXBLOCKS	1024	maximum number of blocks in main memory, if exceed the application is stopped until AUTOFLUSH/ MEM-OVERWRITE/ stop recording trace info
VT_CONFIG_RANK	0	control the process that reads and parses the configuration file



- Avoid rapid-fire dumping trace data into shared filesystems!
- Your fellow cluster users will hate you for it.

Alternatives

- ITAC is deprecated by Intel and will not be further developed (as of 2025)
 - Intel recommends VTune as a replacement, but this is not competitive
- Other tools with similar functionality
 - **Vampir** (commercial, scalable) <https://vampir.eu/>
 - **Scalasca** (for highly scalable programs, no trace view) <https://www.scalasca.org/>
 - **Paraver** <https://tools.bsc.es/paraver>
- Jumpshot
Don't even bother.