



IBM Systems & Technology Group

The IBM High Performance Computing Toolkit on BlueGene/L

Kirk E Jordan, Ph.D.

Strategic Growth Business/Deep Computing
Systems & Technology Group

kjordan@us.ibm.com

Deep Computing

Collaborators:

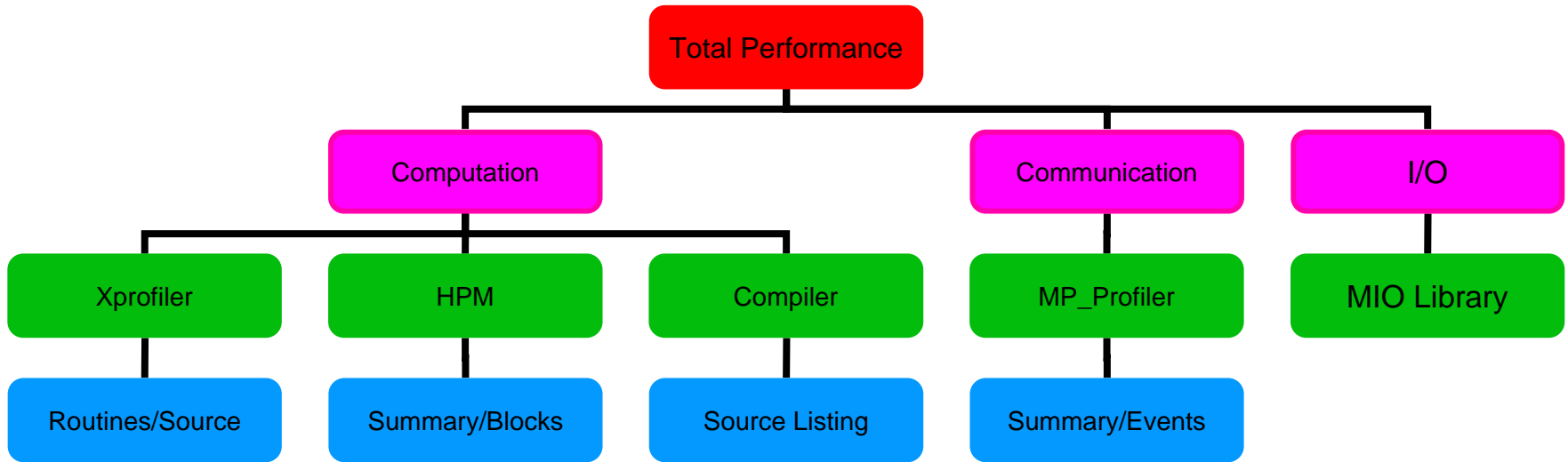
I-Hsin Chung - ACTC

Bob Walkup – Physical Sciences

Outline

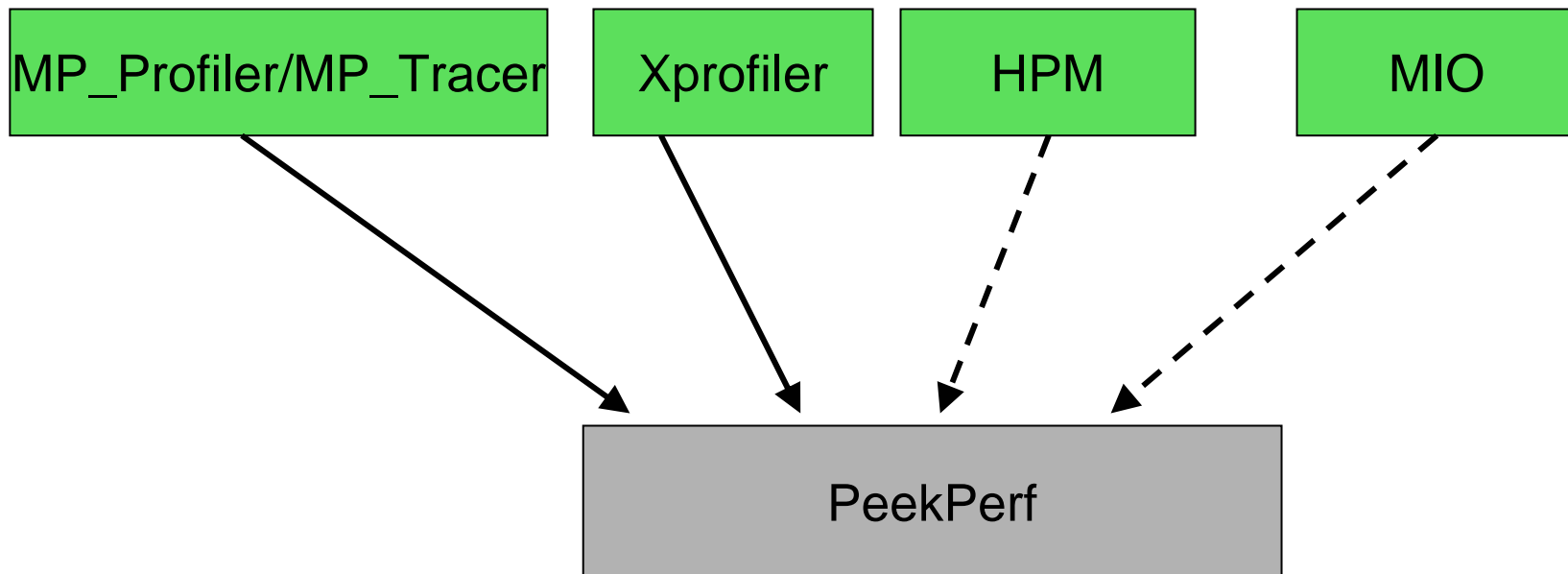
- Performance Decision tree
- IBM High Performance Computing Toolkit
 - MPI performance: MP_Profiler
 - CPU performance: Xprofiler, HPM
 - Modular I/O: MIO
 - Visualization and analysis: PeekPerf
- Challenges
- Future Work

Performance Decision Tree



IBM High Performance Computing Toolkit on BG/L

- MPI performance: MP_Profiler, MP_Tracer
- CPU performance: Xprofiler, HPM
- Visualization and analysis: PeekPerf
- Modular I/O: MIO



Message-Passing Performance:

- **MP_Profiler Library**
 - Captures “summary” data for MPI calls
 - Source code traceback
 - User **MUST** call MPI_Finalize() in order to get output files.
 - No changes to source code
 - MUST compile with `-g` to obtain source line number information
- **MP_Tracer Library**
 - Captures “timestamped” data for MPI calls
 - Source traceback

Compiling and Linking Example

BGL=/bgl/BlueLight/ppcfloor

CC=\$(BGL)/blrts-gnu/powerpc-bgl-blrts-gnu/bin/gcc

CFLAG= -I \$(BGL)/bglsys/include

MPI_LIB= -L \$(BGL)/bglsys/lib -lmpich.rts -lmsglayer.rts -lrts.rts -ldevices.rts

TRACE_LIB= -L \$(MP_PROFILER) -lmpitrace.rts

BINUTILS_LIB= -L \$(BINUTILS) -lbfd -liberty

target: source.c

\$(CC) -o \$@ \$< \$(CFLAG) \$(TRACE_LIB) \$(MPI_LIB) \$(BINUTIL_LIB)



\$(TRACE_LIB) has to precede \$(MPI_LIB)

MP_Profiler Summary Output

```
-----  
MPI Routine           #calls      avg. bytes      time(sec)  
-----  
MPI_Comm_size         3             0.0             0.000  
MPI_Comm_rank        12994         0.0             0.016  
MPI_Send              19575        11166.9         13.490  
MPI_Isend             910791       5804.2          9.216  
MPI_Recv             138173       2767.9          73.835  
MPI_Irecv            784936       15891.6         2.407  
MPI_Sendrecv         894809        352.0           88.705  
MPI_Wait             1537375      0.0            288.049  
MPI_Waitall           44042        0.0            25.312  
MPI_Bcast             464          41936.8         3.272  
MPI_Barrier           1312         0.0            34.206  
MPI_Gather            68           16399.1         2.680  
MPI_Scatter           6            17237.3         0.532  
-----
```

```
total communication time = 770.424 seconds.  
total elapsed time       = 1168.662 seconds.  
user cpu time            = 1160.960 seconds.  
system time              = 0.620 seconds.  
maximum memory size     = 68364 KBytes.
```

```
To check load balance : grep "total comm" mpi_profile.*
```

MP_Profiler Sample Call Graph Output

```
communication time = 143.940 sec, parent = gwrrloc
  MPI Routine      #calls      time(sec)
  MPI_Barrier      2311        143.734
  MPI_Gatherv      2311        0.206
communication time = 137.823 sec, parent = f2drecv
  MPI Routine      #calls      time(sec)
  MPI_Recv         91959      137.823
communication time = 108.960 sec, parent = puttsf
  MPI Routine      #calls      time(sec)
  MPI_Barrier      23607      106.821
  MPI_Gatherv      23607      2.139
communication time = 94.435 sec, parent = fft_tri_recv
  MPI Routine      #calls      time(sec)
  MPI_Recv         6378       94.435
communication time = 83.836 sec, parent = fft2drecv
  MPI Routine      #calls      time(sec)
  MPI_Recv         93003      83.836
```

MP_Profiler Output with Peekperf

The screenshot displays the IBM ACTC PeekPerf: Main Window. The main window shows a list of MPI operations with their call counts and wall clock times. A metric browser window is open, showing a detailed view of the MPI_Send_130 metric.

Metric Browser: MPI_Send_130

Task	Message Size	Count	WallClock [Max]	Transferred Bytes	Call Count [Max]	WallClock
0	(8) 16K ... 64K	1920	0.513883	2.21184e+07	1920	0.513883
1	(8) 16K ... 64K	2880	0.549085	3.31776e+07	2880	0.549085
2	(8) 16K ... 64K	2880	0.551206	3.31776e+07	2880	0.551206
3	(8) 16K ... 64K	1920	0.516694	2.21184e+07	1920	0.516694
4	(8) 16K ... 64K	2880	0.632482	3.31776e+07	2880	0.632482
5	(8) 16K ... 64K	3840	0.654542	4.42368e+07	3840	0.654542
6	(8) 16K ... 64K	3840	0.651453	4.42368e+07	3840	0.651453
7	(8) 16K ... 64K	2880	0.625413	3.31776e+07	2880	0.625413
8	(8) 16K ... 64K	2880	0.593683	3.31776e+07	2880	0.593683
9	(8) 16K ... 64K	3840	0.643496	4.42368e+07	3840	0.643496
10	(8) 16K ... 64K	3840	0.640881	4.42368e+07	3840	0.640881
11	(8) 16K ... 64K	2880	0.589392	3.31776e+07	2880	0.589392
12	(8) 16K ... 64K	2880	0.553126	3.31776e+07	2880	0.553126

The main window also shows a list of MPI operations with their call counts and wall clock times. The **MPI_Send_130** operation is highlighted in blue.

```

+ barrier_sync (mpi_stuff.f)
+ bcast_int (mpi_stuff.f)
+ bcast_real (mpi_stuff.f)
+ global_int_sum (mpi_stuff.f)
+ global_real_max (mpi_stuff.f)
+ global_real_sum (mpi_stuff.f)
+ pmpi_allreduce (allreducef.c)
+ pmpi_barrier (barrierf.c)
+ pmpi_bcast (bcastf.c)
+ pmpi_comm_rank (comm_rankf.c)
+ pmpi_comm_size (comm_sizef.c)
+ pmpi_recv (recvf.c)
+ pmpi_send (sendf.c)
+ rcv_real (mpi_stuff.f)
+ snd_real (mpi_stuff.f)
  MPI_Send_130 3840 0.6
+ SUMMARY 0 0
+ task_init (mpi_stuff.f) 0 0
    
```

The code editor shows the following code:

```

return
end

subroutine rcv_real(orig, value, size, ta
implicit none
    
```

Measuring Communication Performance

The MPI standard provides alternate entry points for timing:

```
MPI_Send(...) {  
    Start_timer();  
    PMPI_Send(...);  
    Stop_timer();  
    Log_the_event();  
}
```

Implement a scalable approach that can support $>10^{**5}$ MPI tasks.

Two modes: summary and event-logging.

MP_Profiler : Summary Mode

- Low overhead : <0.1 microsecond per call
- Low data volume : save just a few small text files
- Make use of the parallelism to do data collection and analysis
 - Save details for MPI ranks with minimum, maximum, and median communication times.
 - Collect total communication time for all ranks in one text file.
 - Collect data for all hardware counter events in one shot.
 - Automatically filter other profile data from gprof etc.

MPI Timing Summary

MPI Rank 0 out of 1024.

elapsed time from clock-cycles using freq = 700.0 MHz.

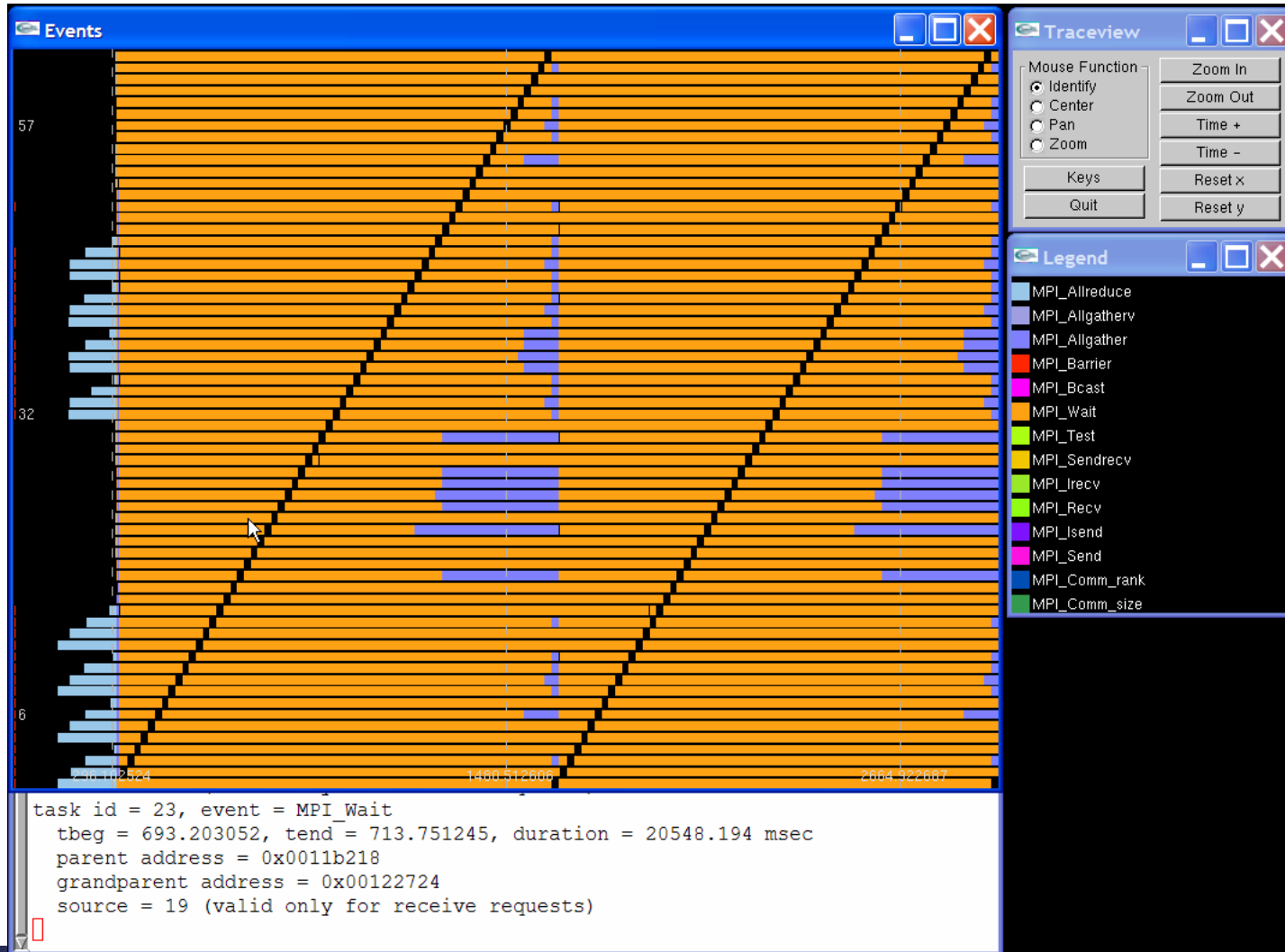
total communication time = 4.943 seconds.
total elapsed time = 43.113 seconds.
top of the heap address = 158.305 MBytes.

MPI Routine	#calls	avg. bytes	time(sec)
MPI_Comm_size	2	0.0	0.000
MPI_Comm_rank	2	0.0	0.000
MPI_Isend	171	765163.0	0.001
MPI_Irecv	174	757120.0	0.000
MPI_Wait	340	0.0	1.912
MPI_Barrier	11	0.0	0.038
MPI_Allreduce	11	16.7	2.992

MP_Profiler : Event Logging

- Visualize the time-sequence of MPI events.
- Preserve the connection between MPI events and source code.
- Use a lightweight fast viewer based on OpenGL.
- Keep the data volume manageable.
- Limit event logging to a subset of MPI ranks, and/or to a subset of the overall simulation.

MPI Events for Enzo - Original



Enzo : Original Code

For each pair of regions

 conditionally call `MPI_Test(...)` to check for outstanding send requests

 if regions overlap

 call `MPI_Isend(send_request,...)` to “send” data

For each pair of regions

 if regions overlap

 call `MPI_Irecv(recv_request)` to receive boundary data

 call `MPI_Wait(recv_request,...)`

Enzo : Modified Code

For each pair of regions

 conditionally call `MPI_Test(...)` to check for outstanding send requests

 if regions overlap

 call `MPI_Isend(send_request,...)` to “send” data

call `MPI_Barrier(...)`

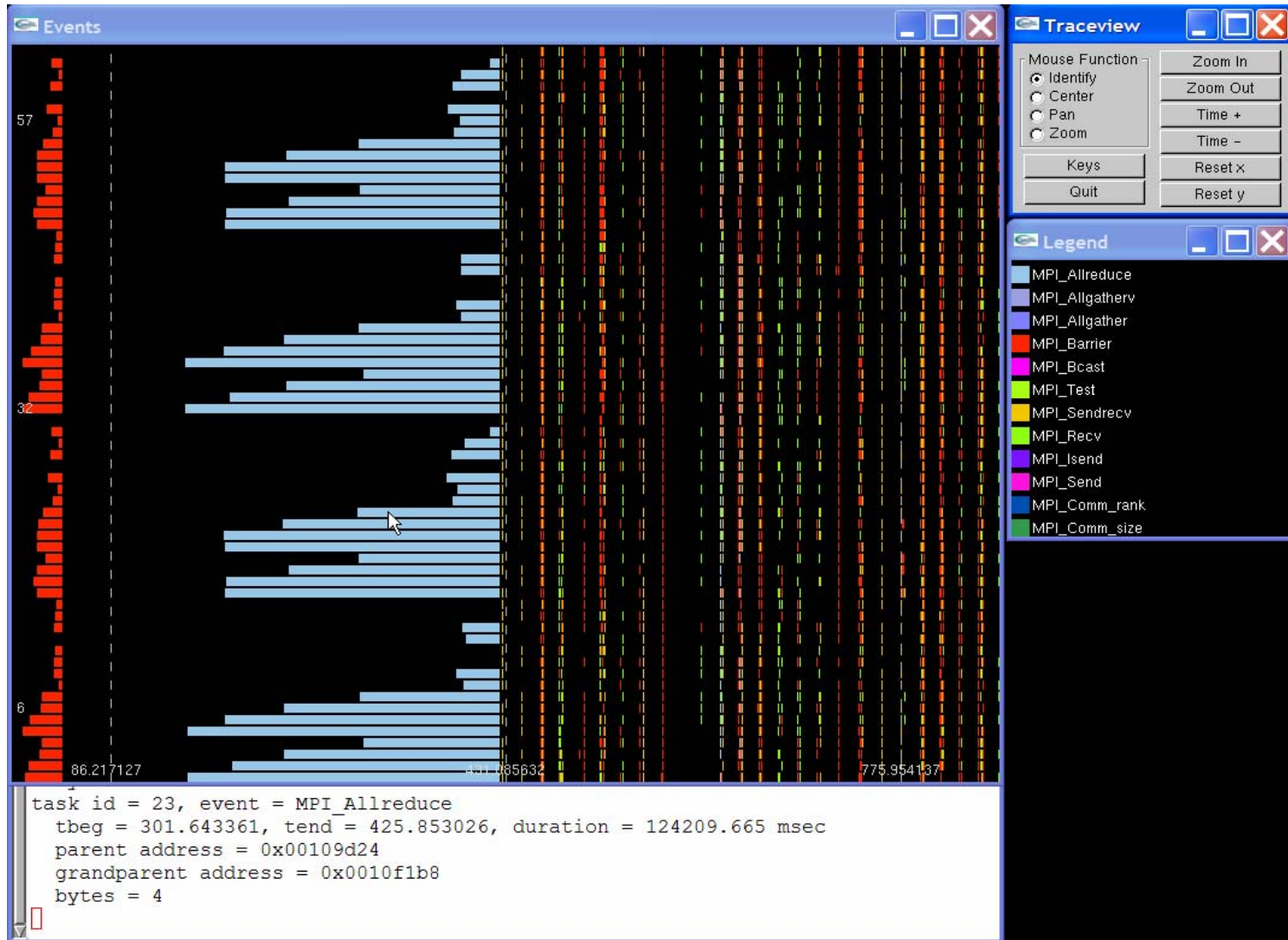
For each pair of regions

 if regions overlap

 call `MPI_Irecv(recv_request)` to receive boundary data

 call `MPI_Wait(recv_request,...)`

MPI Events for Enzo - Tuned



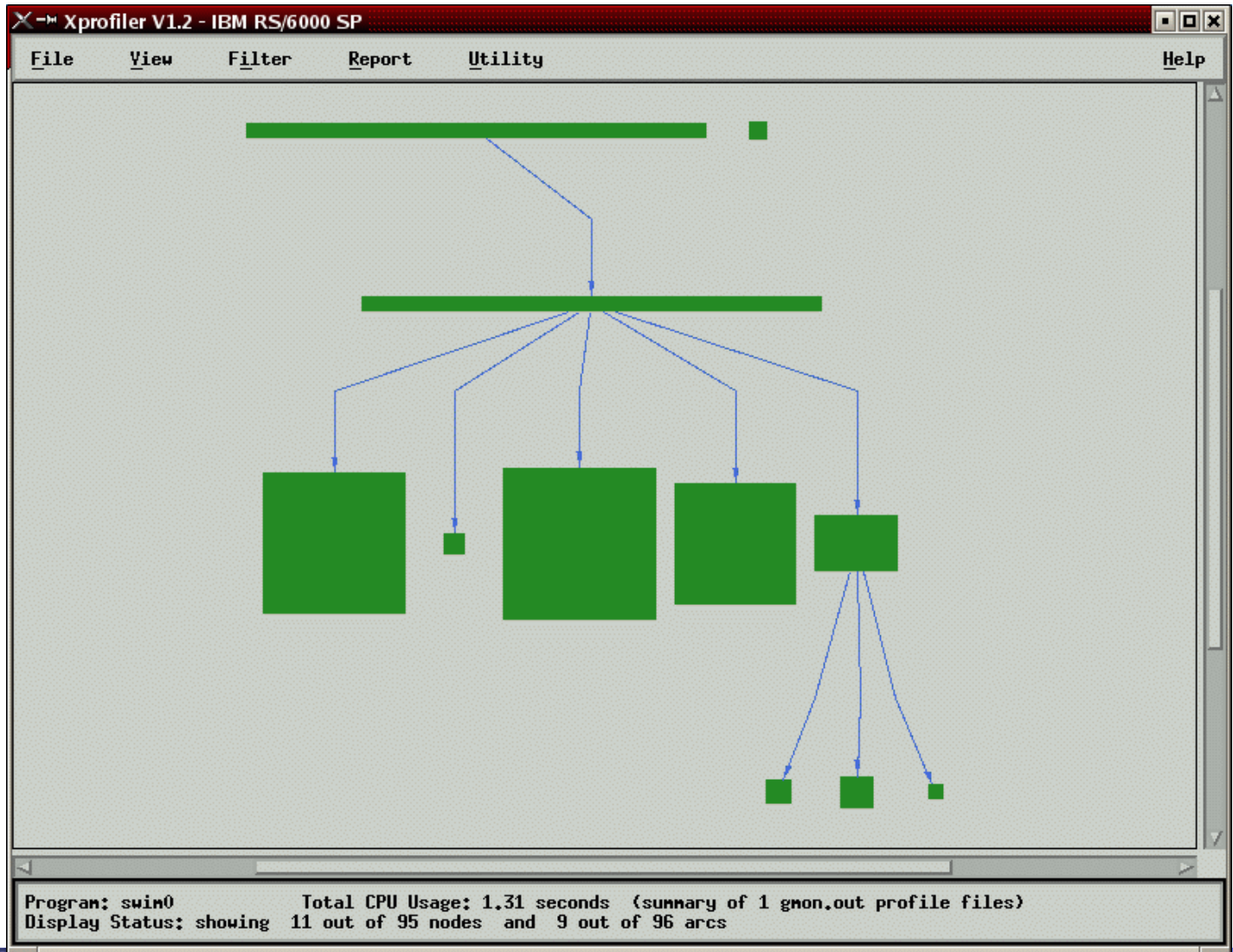
Xprofiler

- CPU profiling tool similar to gprof
- Can be used to profile both serial and parallel applications
- Use procedure-profiling information to construct a graphical display of the functions within an application
- Provide quick access to the profiled data and helps users identify functions that are the most CPU-intensive
- Based on sampling (support from both compiler and kernel)
- Charge execution time to source lines and show disassembly code

Running Xprofiler

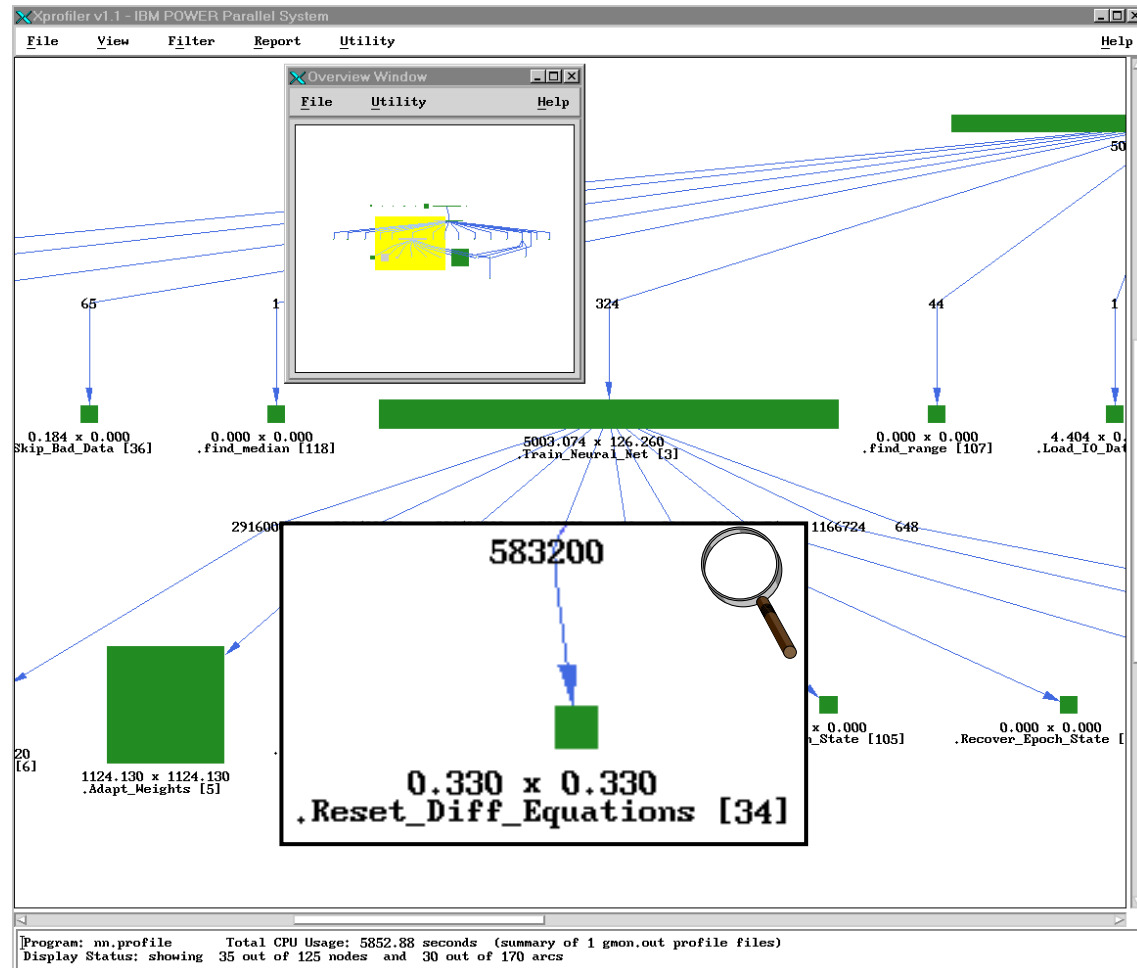
- Compile the program with `-pg`
- Run the program
- `gmon.out` file is generated (MPI applications generate `gmon.out.1, ..., gmon.out.n`)
- Run Xprofiler

Xprofiler - Application View



Xprofiler: Main Display

- Width of a bar: time including called routines
- Height of a bar: time excluding called routines
- Call arrows labeled with number of calls
- Overview window for easy navigation (View → Overview)



Xprofiler: Flat Profile

- Menu **Report** provides usual gprof reports plus some extra ones
 - Flat Profile
 - Call Graph Profile
 - Function Index
 - Function Call Summary
 - Library Statistics

The screenshot shows the 'Flat Profile' window with a table of function statistics. The table has columns for %time, cumulative seconds, self seconds, calls, self ms/call, total ms/call, and name. The top row is highlighted with a magnifying glass.

%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
62.9	15.64	15.64	1	15640.00	15650.00	.main [1]
33.9	24.08	8.44	10	844.00	844.00	.thsub [3]
2.7	24.75	0.67				.datb [5]
0.2	24.81	0.06				.dtrca [6]
0.2	24.85	0.04				.durand [7]
0.0	24.86	0.01	28	0.36	0.36	.fwrite_unlocked [9]
0.0	24.87	0.01				.dgetmo [12]
0.0	24.87	0.00	55	0.00	0.00	.leftmost [13]
0.0	24.87	0.00	28	0.00	0.36	.fwrite [8]
0.0	24.87	0.00	28	0.00	0.00	.memchr [19]
0.0	24.87	0.00	16	0.00	0.00	.rightmost [20]
0.0	24.87	0.00	10	0.00	0.00	.mtdsqmm [21]
0.0	24.87	0.00	10	0.00	0.00	.splint [22]
0.0	24.87	0.00	10	0.00	0.00	.syncthread [23]
0.0	24.87	0.00	9	0.00	1.11	._doprint [10]
0.0	24.87	0.00	9	0.00	0.00	._xflsbuf [24]
0.0	24.87	0.00	9	0.00	0.00	._xwrite [25]
0.0	24.87	0.00	9	0.00	1.11	.printf [11]
0.0	24.87	0.00	9	0.00	0.00	.time_base_to_time [26]

Search Engine: (regular expressions supported)

Xprofiler: Source Code Window

- Source code window displays source code with time profile (in ticks=.01 sec)
- Access
 - Select function in main display
 - → context menu
 - Select function in flat profile
 - → Code Display
 - → Show Source Code

line	no. ticks per line	source code
202		/*-----*/
203		/* use 2x-unrolling of the outer two loops */
204		/*-----*/
205	4	for (i=i0; i<i0+is-1; i+=2)
206		{
207	8	for (j=j0; j<j0+js-1; j+=2)
208		{
209	1	t11 = c[i*n+j];
210	5	t12 = c[i*n+j+1];
211	5	t21 = c[(i+1)*n+j];
212	19	t22 = c[(i+1)*n+(j+1)];
213		for (k=k0; k<k0+ks; k++)
217	229	t21 = t21 + a[(i+1)*n+k]*bt[j*n+k];
218	144	t22 = t22 + a[(i+1)*n+k]*bt[(j+1)*n+k];
219		}
220	7	c[i*n+j] = t11;
221		c[i*n+j+1] = t12;
222	3	c[(i+1)*n+j] = t21;
223	5	c[(i+1)*n+(j+1)] = t22;
224		}
225		for (j=j; j<j0+js; j++)
226		{
227		t11 = c[i*n+j];
228		t21 = c[(i+1)*n+j];
229		for (k=k0; k<k0+ks; k++)
230		{
231		t11 = t11 + a[i*n+k]*bt[j*n+k];
232		t21 = t21 + a[(i+1)*n+k]*bt[j*n+k];
233		}
234		c[i*n+j] = t11;
235		c[(i+1)*n+j] = t21;
236		}
237		}

Search Engine: (regular expressions supported)

thsub

Xprofiler - Disassembler Code

Disassembler Code for .calc3 [3]

address	no. ticks per instr.	instruction	assembler code	source code
10002E18	81	FCC4287C	fnms 6, 4, 1, 5	
10002E1C	64	CCF70008	lfd 7, 0x8(23)	POLD(I, J) = P(I, J)+ALPHA*(PNEW(I, J)-
10002E20	187	C90C0008	lfd 8, 0x8(12)	
10002E24	53	C9750008	lfd 11, 0x8(21)	UOLD(I, J) = U(I, J)+ALPHA*(UNEW(I, J)-
10002E28	89	FD63582A	fa 11, 3, 11	
10002E2C	63	FD28387C	fnms 9, 8, 1, 7	POLD(I, J) = P(I, J)+ALPHA*(PNEW(I, J)-
10002E30	4	DD5B0008	stfdu 10, 0x8(27)	U(I, J) = UNEW(I, J)
10002E34		C9540008	lfd 10, 0x8(20)	VOLD(I, J) = V(I, J)+ALPHA*(VNEW(I, J)-
10002E38	113	FCCA302A	fa 6, 10, 6	
10002E3C	27	C8760008	lfd 3, 0x8(22)	POLD(I, J) = P(I, J)+ALPHA*(PNEW(I, J)-
10002E40	87	FD8012FA	fma 12, 0, 11, 2	UOLD(I, J) = U(I, J)+ALPHA*(UNEW(I, J)-
10002E44	35	DCB90008	stfdu 5, 0x8(25)	V(I, J) = VNEW(I, J)
10002E48	4	FC63482A	fa 3, 3, 9	POLD(I, J) = P(I, J)+ALPHA*(PNEW(I, J)-
10002E4C	12	CD5A0008	lfd 10, 0x8(26)	UOLD(I, J) = U(I, J)+ALPHA*(UNEW(I, J)-
10002E50	62	FCC021BA	fma 6, 0, 6, 4	VOLD(I, J) = V(I, J)+ALPHA*(VNEW(I, J)-
10002E54	36	C85B0008	lfd 2, 0x8(27)	UOLD(I, J) = U(I, J)+ALPHA*(UNEW(I, J)-
10002E58	244	DCEC0008	stfdu 7, 0x8(12)	P(I, J) = PNEW(I, J)
10002E5C	28	FD0040FA	fma 8, 0, 3, 8	POLD(I, J) = P(I, J)+ALPHA*(PNEW(I, J)-
10002E60		C8990008	lfd 4, 0x8(25)	VOLD(I, J) = V(I, J)+ALPHA*(VNEW(I, J)-
10002E64	316	DCD40008	stfdu 6, 0x8(20)	
10002E68	29	FC62507C	fnms 3, 2, 1, 10	UOLD(I, J) = U(I, J)+ALPHA*(UNEW(I, J)-

Search Engine: (regular expressions supported)

Hardware Performance Monitor (HPM)

- Provides comprehensive reports of events that are critical to performance on IBM systems.
- Gather critical hardware performance metrics, e.g.
 - Number of misses on all cache levels
 - Number of floating point instructions executed
 - Number of instruction loads that cause TLB misses
- Helps to identify and eliminate performance bottlenecks.

LIBHPM

- Instrumentation library
- Provides performance information for instrumented program sections
- Supports multiple (nested) instrumentation sections
- Multiple sections may have the same ID
- Run-time performance information collection
- Based on `bgl_perfctr` layer – can be eliminated in BG/P

Event Sets

- 16 sets (0-15); 328 events
- Information for
 - Time
 - FPU (0,1)
 - L3 memory
 - Processing Unit (0,1)
 - Tree network
 - Torus network
- For detailed names and descriptions: [event_sets.txt](#)

Functions

- `hpmInit(taskID, progName) / f_hpmInit(taskID, progName)`
 - `taskID` is an integer value indicating the node ID.
 - `progName` is a string with the program name.

- `hpmStart(instID, label) / f_hpmStart(instID, label)`
 - `instID` is the instrumented section ID. It should be > 0 and ≤ 100 (can be overridden)
 - `Label` is a string containing a label, which is displayed by PeekPerf

- `hpmStop(instID) / f_hpmStop(instID)`
 - For each call to `hpmStart`, there should be a corresponding call to `hpmStop` with matching `instID`

- `hpmTerminate(taskID) / f_hpmTerminate(taskID)`
 - This function will generate the output. If the program exits without calling `hpmTerminate`, no performance information will be generated.

Functions (continued)

- `hpmGetTimeAndCounters(numCounters, time, values)`
`/ f_GetTimeAndCounters (numCounters, time, values)`
 - returns the time in seconds and counts since the call to `hpmInit`.
 - `numCounters`: integer indicating the number of counters to be accessed.
 - `time`: double precision float
 - `values`: “long long” vector of size “`numCounters`”.

- `hpmGetCounters(values) / f_hpmGetCounters (values)`
 - Similar to `hpmGetTimeAndCounters`
 - only returns the total counts since the call to `hpmInit`

Example of Instrumented Code

■ C / C++

declaration:

```
#include "libhpm.h"
```

use:

```
hpmInit( taskID, "my program" );  
hpmStart( 1, "outer call" );  
do_work();  
hpmStart( 2, "computing meaning of  
          life" );  
do_more_work();  
hpmStop( 2 );  
hpmStop( 1 );  
hpmTerminate( taskID );
```

■ Fortran

declaration:

```
#include "f_hpm.h"
```

use:

```
call f_hpminit( taskID, "my program" )  
call f_hpmstart( 1, "Do Loop" )  
do ...  
  call do_work()  
  call f_hpmstart( 5, "computing meaning  
                  of life" );  
  call do_more_work();  
  call f_hpmstop( 5 );  
end do  
call f_hpmstop( 1 )  
call f_hpmterminate( taskID )
```

Hpmcount Example

```
bash-2.05a$ hpm_count swim
..... // program output
hpmcount (V 2.5.4) summary
Execution time (wall clock time)                : 7.378159 seconds
##### Resource Usage Statistics #####
Total amount of time in user mode                : 0.010000 seconds
Average shared memory use in text segment        : 672 Kbytes*sec
.....
PM_FPU_FDIV (FPU executed FDIV instruction)      : 0
PM_FPU_FMA (FPU executed multiply-add instruction) : 0
PM_CYC (Processor cycles)                        : 54331072
PM_FPU_STF (FPU executed store instruction)     : 2172
PM_INST_CMPL (Instructions completed)           : 17928229
Utilization rate                                 : 0.446 %
Total load and store operations                  : 0.004 M
MIPS                                              : 2.140
Instructions per cycle                            : 0.330
```

HPM Visualization Using PeekPerf

hpmviz

File

swim_omp | swim_omp.f | calc1.f | calc2.f | calc3.f

Label	ExcSec	IncSec	Count
Loop 300	4.572	4.572	2398
Loop 200	4.203	4.203	2400
Loop 100	3.071	3.071	2400
Calc3	1.838	6.813	2398
Calc2	1.013	5.632	2400

```

*   VOLD(N1,N2), POLD(N1,N2),
2   CU(N1,N2), CV(N1,N2),
*   Z(N1,N2), H(N1,N2), PSI(N1,N2)
C
COMMON /CONS/ DT,TDT,DX,DY,A,ALPHA,ITMAX,MPRINT,
1   NP1,EL,PI,TPI,DI,DJ,PCF
integer ierr
                    
```

Metric Browser: Loop 300

Node	Thread	Count	ExcSec	IncSec	U time	Use rate	(M) LS	MIPS	HW FP/Cyc	Instr/LS	M Flips	IpC	Mflip/s	WFlips	Wflip/s
0	3	2398	4.539	4.539	3.923	86.425	590.056	291.855	0.116	2.245	589.86	0.26	129.947	589.86	129.9
0	0	2398	4.572	4.572	4.378	95.763	608.414	263.277	0.107	1.978	608.234	0.211	133.037	608.234	133.0
0	2	2398	4.549	4.549	4.366	95.979	590.019	255.241	0.104	1.968	589.838	0.205	129.663	589.838	129.6
0	1	2398	4.547	4.547	4.308	94.759	590.024	259.19	0.105	1.997	589.837	0.21	129.728	589.837	129.7
1	2	2398	4.534	4.534	4.398	96.999	590.044	253.123	0.103	1.945	589.856	0.201	130.088	589.856	130.0
1	1	2398	4.528	4.528	3.942	87.069	589.983	286.058	0.115	2.195	589.807	0.253	130.263	589.807	130.2
1	0	2398	4.547	4.547	3.766	82.828	608.434	308.065	0.124	2.302	608.244	0.286	133.762	608.244	133.7
1	3	2398	4.523	4.523	3.537	78.198	589.962	317.346	0.128	2.433	589.781	0.312	130.4	589.781	130.4
2	0	2398	4.538	4.538	3.777	83.218	608.448	312.01	0.124	2.327	608.262	0.288	134.029	608.262	134.0
2	2	2398	4.522	4.522	4.313	95.364	590.033	257.962	0.105	1.977	589.86	0.208	130.431	589.86	130.4
2	3	2398	4.52	4.52	4.307	95.285	589.985	258.863	0.105	1.983	589.806	0.209	130.492	589.806	130.4
2	1	2398	4.52	4.52	4.35	96.222	589.943	255.814	0.104	1.96	589.767	0.205	130.466	589.767	130.4
3	3	2398	4.487	4.487	4.193	93.453	571.551	259.827	0.105	2.04	571.374	0.214	127.352	571.374	127.3
3	1	2398	4.502	4.502	4.365	96.953	589.937	254.196	0.104	1.94	589.763	0.202	131.003	589.763	131.0
3	2	2398	4.483	4.483	4.139	92.33	571.556	263.864	0.106	2.07	571.38	0.22	127.445	571.38	127.4
3	0	2398	4.506	4.506	3.927	87.154	590.044	290.852	0.116	2.221	589.856	0.257	130.901	589.856	130.9

```

V(I,J) = VNEW(I,J)
P(I,J) = PNEW(I,J)
300 CONTINUE
call f_hpmrstop( 30+omp_get_thread_num())
                    
```

Deep Computing

Environment Flags

- **HPM_EVENT_SET**
 - Select the event set to be recorded
 - Integer (0 – 15)
- **HPM_NUM_INST_PTS**
 - Overwrite the default of 100 instrumentation sections in the app.
 - Integer value > 0
- **HPM_WITH_MEASUREMENT_ERROR**
 - Deactivate the procedure that removes measurement errors.
 - True or False (0 or 1).
- **HPM_OUTPUT_NAME**
 - Define an output file name different from the default.
 - String
- **HPM_VIZ_OUTPUT**
 - Indicate if “.viz” file (for input to PeekPerf) should be generated or not.
 - True or False (0 or 1).
- **HPM_TABLE_OUTPUT**
 - Indicate table text file should be generated or not.
 - True or False (0 or 1).

Modular I/O (MIO)

- Addresses the need of application-level optimization for I/O.
- Analyze and tune I/O at the application level
 - For example, when an application exhibits the I/O pattern of sequential reading of large files
 - MIO
 - Detects the behavior
 - Invokes its asynchronous prefetching module to prefetch user data.
- Planned Integration into HPC Toolkit with PeekPerf capabilities
 - Source code traceback
 - Future capability for dynamic I/O instrumentation

Challenges

- Hardware
 - Different performance counter set
- Scalability
 - 64k nodes
 - Tracking communications for each pair:
 - $(65,536)^2 \times 48$ bytes = 200 GB
 - Number of processes and events vs. number of screen pixels

Future work

- **MP_Profiler**
 - Scalability
- **Xprofiler**
 - Improved GUI
 - Integration with other platforms
- **HPM**
 - Useful derived metrics and verifications
 - Hpmcount / Hpmstat ?
 - Integration with other platforms

- **Next target**
 - Modular I/O: MIO