The challenge of adapting HEP physics software applications to run on many-core cpus

CERN, June `10

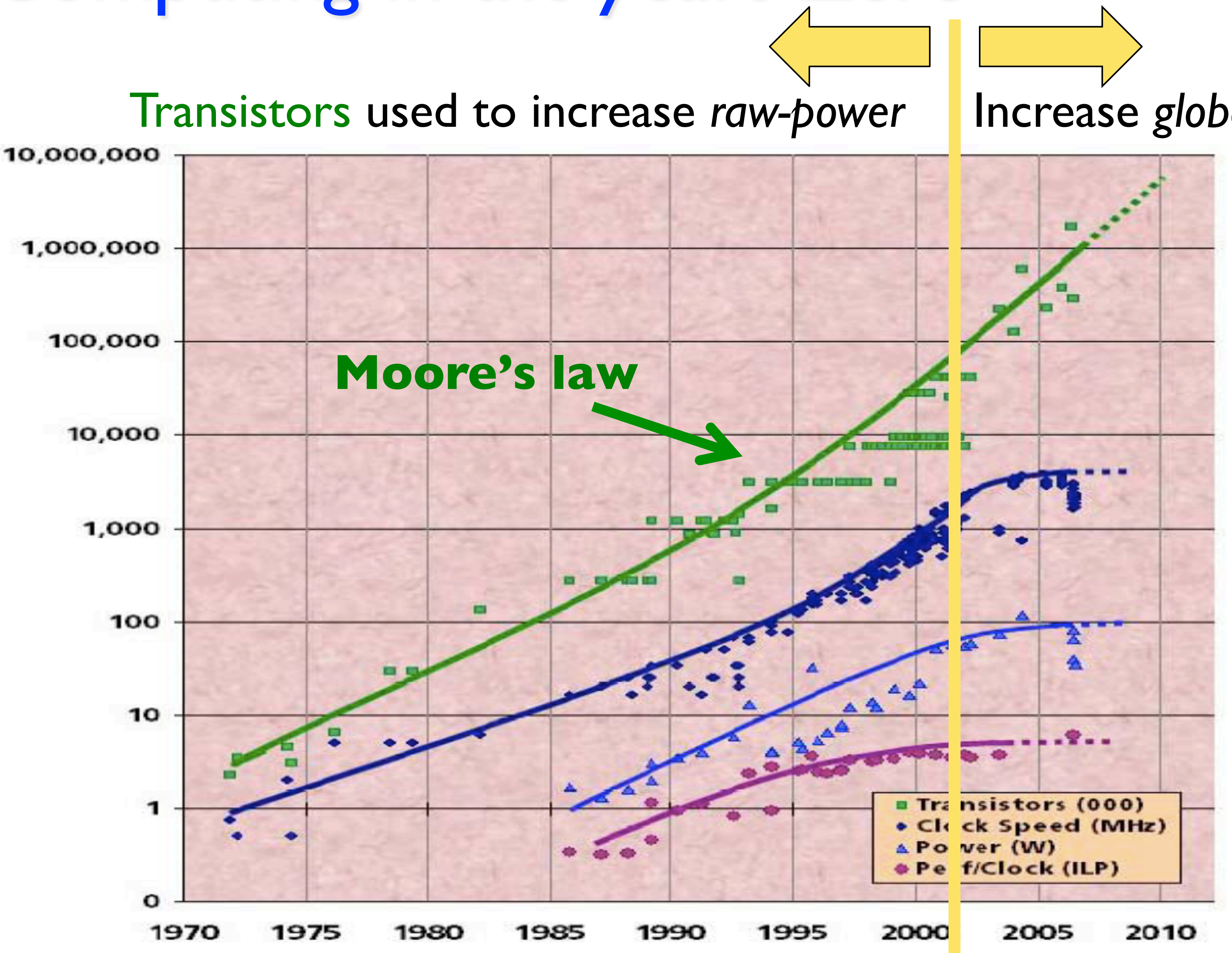
High Performance Computing
for High Energy Physics

Vincenzo Innocente
CERN

MOTIVATIONS

Computing in the years Zero

Transistors used to increase *raw-power* | Increase *global power*

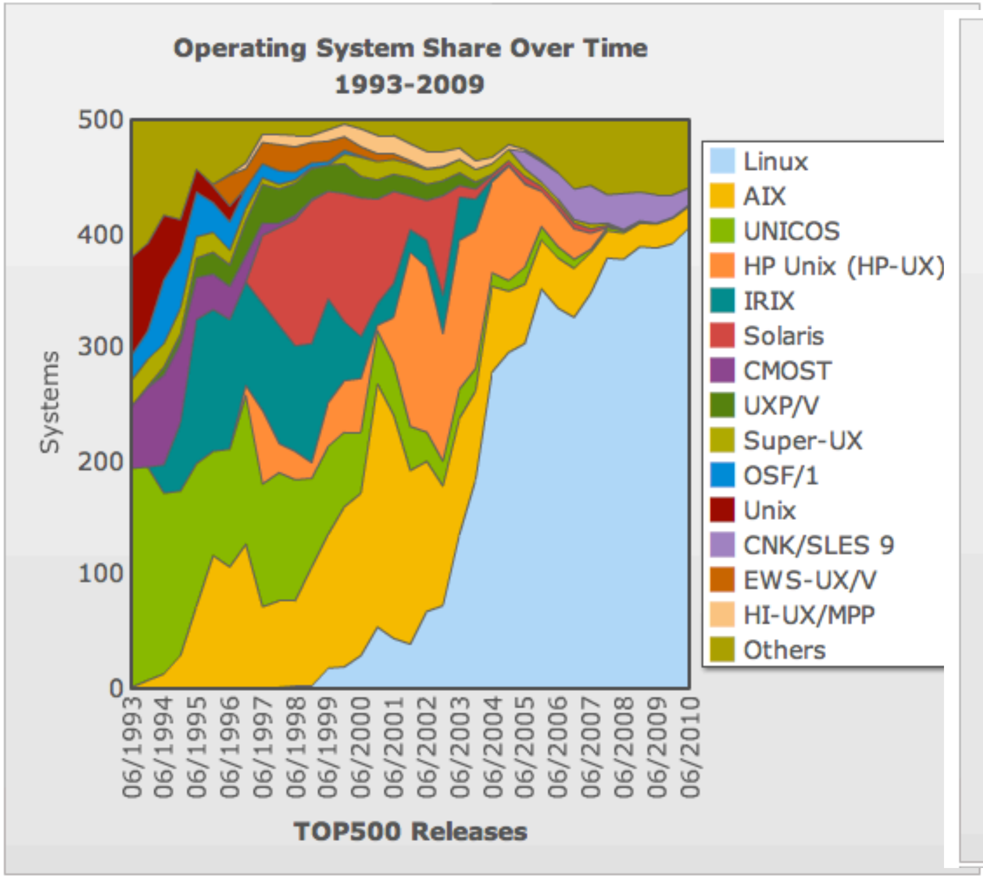


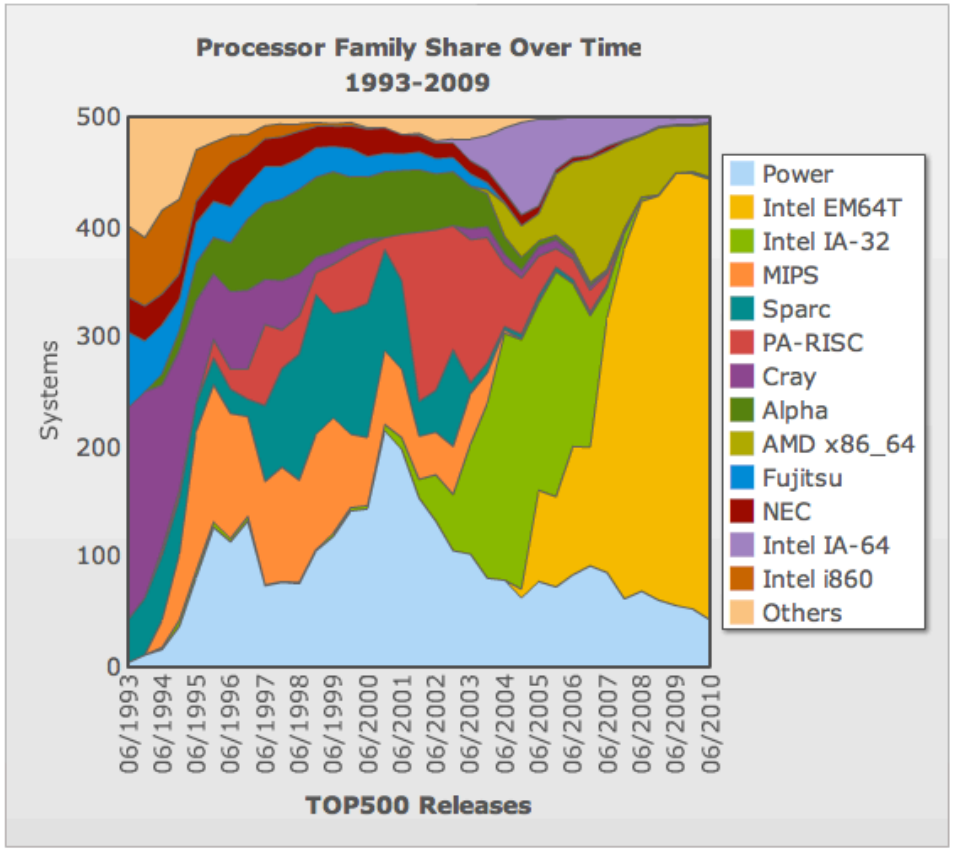
Go Parallel: many-cores!

- A turning point was reached and a new technology emerged: **multicore**
  - » Keep frequency and consumption low
  - » Transistors used for multiple cores on a single chip: 2, 4, 6, 8 cores on a single chip
- Multiple hardware-threads on a single core
  - » simultaneous Multi-Threading (Intel Core i7 2 threads per core (6 cores), Sun UltraSPARC T2 8 threads per core (8 cores))
- Dedicated architectures:
  - » GPGPU: up to 240 threads (NVIDIA, ATI-AMD, Intel MIC)
  - » CELL
  - » FPGA (Reconfigurable computing)

Top 500 1993-2010

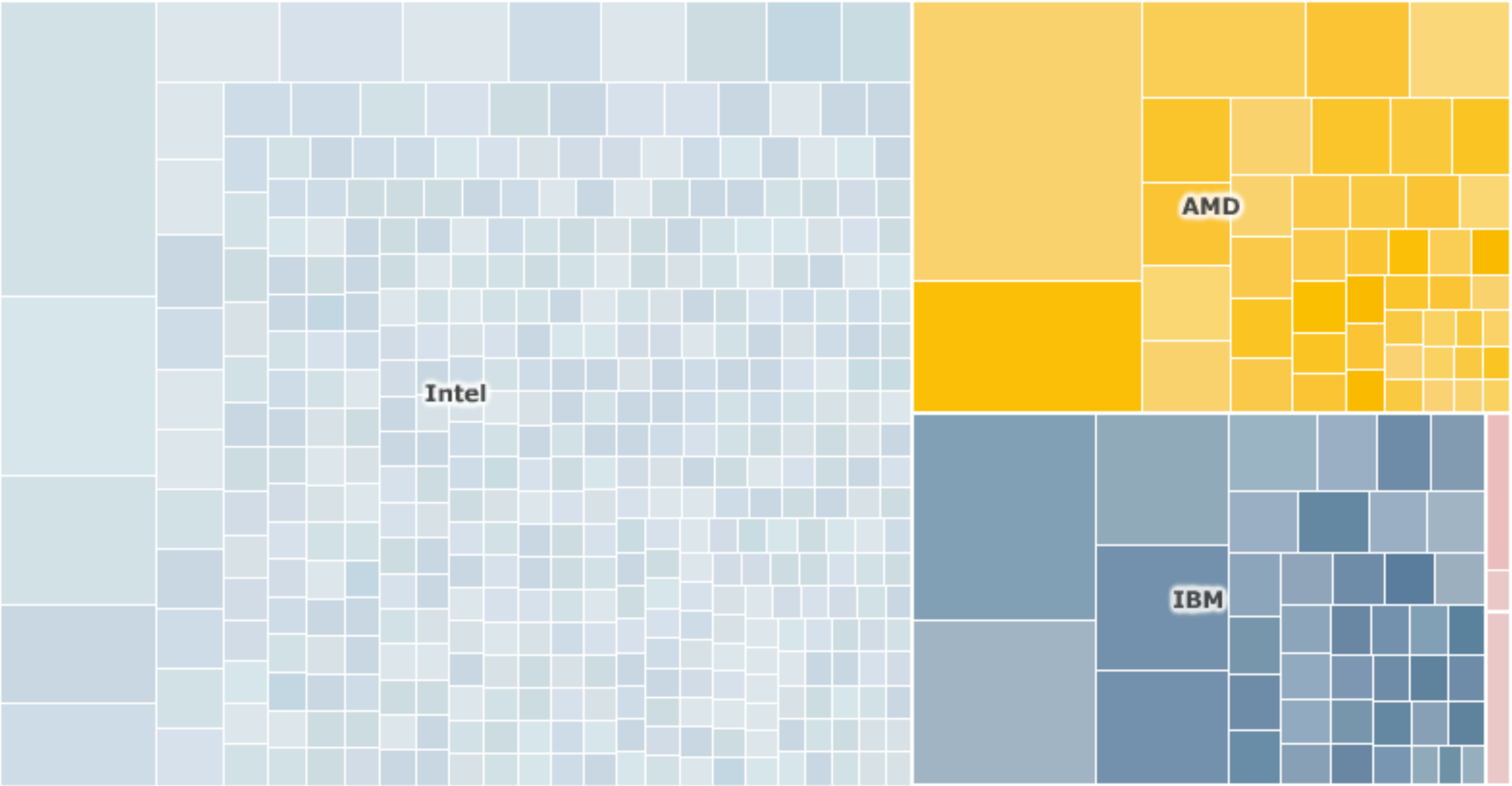
Source http://www.top500.org/





Top 500 in 2010

Source BBC http://news.bbc.co.uk/2/hi/technology/10187248.stm



Moving to a new era

1990

- Many architectures
  - » Evolving fast
- Many OS, Compilers, libraries
  - » optimized to a given architecture
- Stead increase of single processor speed
  - » Faster clock
  - » flexible instruction pipelines
  - » Memory hierarchy
- High level software often unable to exploit all these goodies

2010

- One architecture
  - » Few vendor variants
- One Base Software System
- Little increase in single processor speed
- Opportunity to tune performances of application software
  - » Software specific to Pentium3 still optimal for latest INTEL and AMD cpus

HEP SOFTWARE IN THE MULTICORE ERA

HEP software on multicore: an R&D project *(WP8 in CERN/PH)*

The aim of the WP8 *R&D project is to investigate novel software solutions to efficiently exploit the new multi-core architecture of modern computers in our HEP environment*

Motivation:

industry trend in workstation and "medium range" computing

Activity divided in four "tracks"

- » Technology Tracking & Tools
- » System and core-lib optimization
- » Framework Parallelization
- » Algorithm Optimization and Parallelization

Coordination of activities already on-going in exps, IT, labs

Where are WE?

Experimental HEP is blessed by the natural parallelism of Event processing

- HEP code does not exploit the power of current processors
  - One instruction per cycle at best
  - Little or no use of vector units (SIMD)
  - Poor code locality
  - Abuse of the heap
- Running N jobs on N=8/12 cores still "efficient" but:
  - Memory (and to less extent cpu cycles) wasted in non sharing
    - "static" condition and geometry data
    - I/O buffers
    - Network and disk resources
  - Caches (memory on CPU chip) wasted and trashed
    - L1 cache local per core, L2 and L3 shared
    - Not locality of code and data

This situation is already bad today, will become only worse in future many-cores architecture

Code optimization

- Ample Opportunities for improving code performance
  - » Measure and analyze performance of current LHC physics application software on multi-core architectures
  - » Improve data and code locality (avoid trashing the caches)
  - » Effective use of vector/streaming instruction (SSE, future AVX)
  - » Exploit modern compiler's features (does the work for you!)
- *See Paolo Calafiura's talk @ CHEP09*:
  http://indico.cern.ch/contributionDisplay.py?contribId=517&sessionId=1&confId=35523
- Direct collaboration with INTEL experts established to help analyzing and improve the code

- All this is absolutely necessary, still not sufficient to take full benefits from the modern many-cores architectures
  - » NEED work on the code to have good parallelization

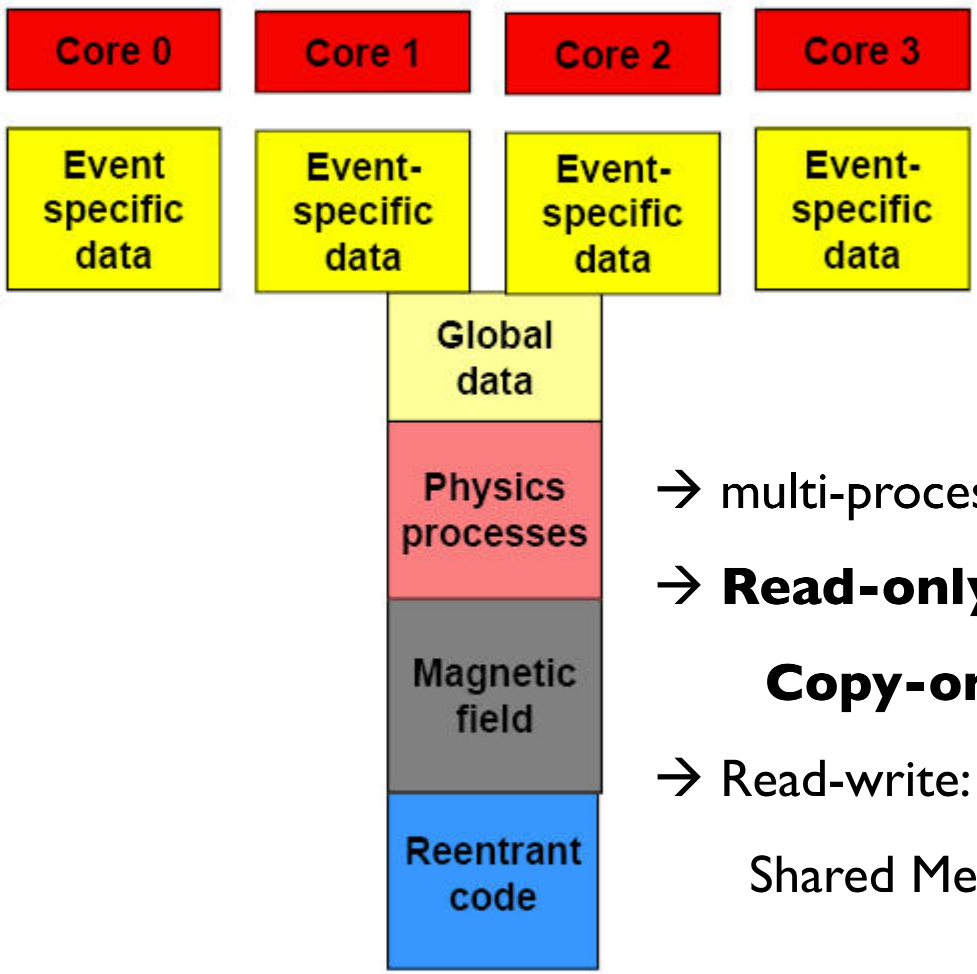
Instrument, measure, improve

- Experiment frameworks (CMSSW, Gaudi, Geant4) instrumented to capture performance counters in specific context (by module, by G4-volume, by G4-particle)
- All experiments, G4, Root successfully reduced memory allocation
- Use of streaming/vector instructions improved float algorithms used in reconstruction by factor 2 (theoretical max is 4)
  - » Promising for double-precision in next generation INTEL/AMD cpus
- Speed-up observed when using auto-vectorization in gcc 4.5
- Work started to improve code locality (reduce instruction cache-misses)

Event parallelism

Opportunity: Reconstruction Memory-Footprint shows large condition data

How to share common data between different process?



→ multi-process vs multi-threaded

→ **Read-only:**

Copy-on-write, Shared Libraries

→ Read-write:

Shared Memory, Sockets, Files

Multithreaded Geant4 (Geant4MT)

- Event-level parallelism to simulate separate events by multiple threads
- Efficiency for future many-core CPUs
- Testing and validation on today's 4-, 8- and 24-core nodes
- Preliminary results available based on testing on `fullCMS bench1.g4`
- Patch parser.c of gcc to output static and global declarations in Geant4 source code and add the "__thread" keyword
- Separate and share read-only data members : Geant4 parameterised geomeries and replicas, Geant4 materials and particles, Geant4 physics tables, etc.
- Custom malloc library to support thread private allocation
- Modified G4Navigator to remove unnecessary updates to G4cout and G4cerr precision (shared variables)

"Multi-core & multi-threading: Tips on how to write "thread-safe" code in Geant4", Xin Dong and Gene Cooperman, *14th Geant4 Users and Collaboration Workshop Search*, `http://indico.cern.ch/sessionDisplay.py?sessionId=68\&slotId=0\&confId=44566#2009-` and `http://indico.cern.ch/conferenceDisplay.py?confId=44566`

Experimental Results on 24-core Intel Xeon 7400 Computer

By segregating read-write data members, large read-only memory chunks are formed. Copy-On-Write does not replicate those read-only chunks. (Geant4MT + COW)

- Separate Processes: No reduction for the memory footprint
- Geant4 + COW: Share geometries (no replica or parameterized geometry)
- Geant4MT + COW: Reduce the memory footprint
- Geant4MT: Reduce the memory footprint

Tested on `fullCMS bench1.g4` with 24 workers and 4000 events per worker (electromagnetics).

Implementation	Total Memory on master	Additional Memory per Worker	Total Memory (master + 24 workers)	Runtime
Separate Processes	250 MB	250 MB	6 GB	4575 s
Original Geant4 + COW	250 MB	70 MB	2G MB	4571 s
Geant4MT + COW	250 MB	20 MB	730 MB	4540 s
Geant4MT 24 threads	250 MB	20 MB	730 MB	4510 s

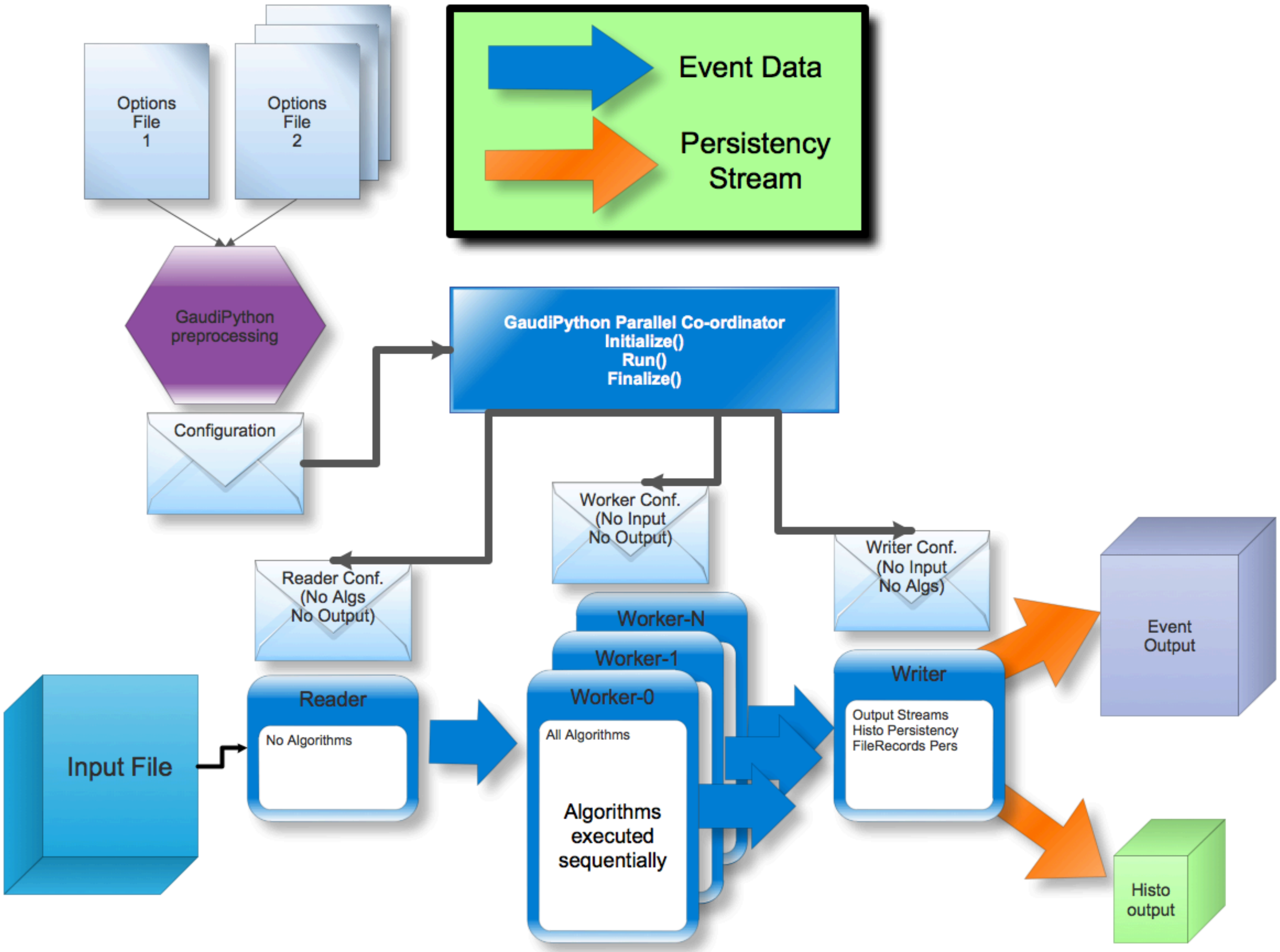
Performance After Output Privatization

Removal of writes to shared G4cout.precision on 4 Intel Xeon 7400 Dunnington

Number of Workers	# Instructions	L3 References	Before Removal		After Removal		
			L3 Misses	CPU Cycles	L3 Misses	Time	Speedup
1	1,598G	87415M	293M	1945G	308M	6547s	1
6	1,598G	87878M	326M	2100G	302M	1087s	6.02
12	1,598G	88713M	456M	3007G	302M	543s	12.06
24	1,599G	88852M	517M	3706G	294M	271s	24.16

Allocator comparison on 4 AMD Opteron 8346 HE

#Wks.	ptmalloc2		ptmalloc3		hoard		tcmalloc		tpmalloc	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	9923s	1	10601s	1	10503s	1	9918s	1	10090s	1
2	4886s	2.03	6397s	1.66	6316s	1.66	4980s	1.99	5024s	2.01
4	2377s	4.17	4108s	2.58	2685s	3.91	2564s	3.87	2504s	4.03
8	1264s	7.85	2345s	4.52	1321s	7.95	1184s	8.37	1248s	8.08
16	797s	12.46	1377s	7.70	691s	15.20	660s	15.02	623s	16.20



GaudiPython parallel

eoin.smith@cern.ch

GaudiPython Parallel

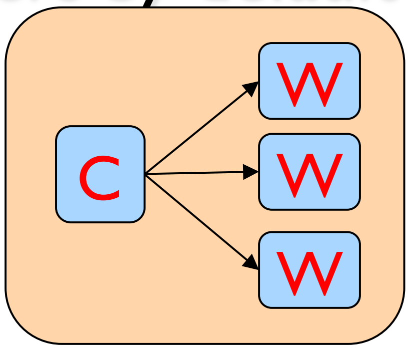
- Reconstruction (Brunel)
  - » FEST-2009-Data.py : 1000 Events
    - From $BRUNELOPTS

Run Type	CPU%	T_elapsed	T_init	T_run	Speedup
Serial		1334	47	1287	1
parallel=5		317	47	280	4.6

- ~1.5s/event
  - Parallel Overhead 3%
  - Speedup Near-Linear

PROOF Lite

- PROOF Lite is a realization of PROOF in 2 tiers
  - The client starts and controls directly the workers
  - Communication goes via UNIX sockets
- No need of daemons:
  - workers are started via a call to ‘system’ and call back the client to establish the connection
- Starts N_{CPU} workers by default

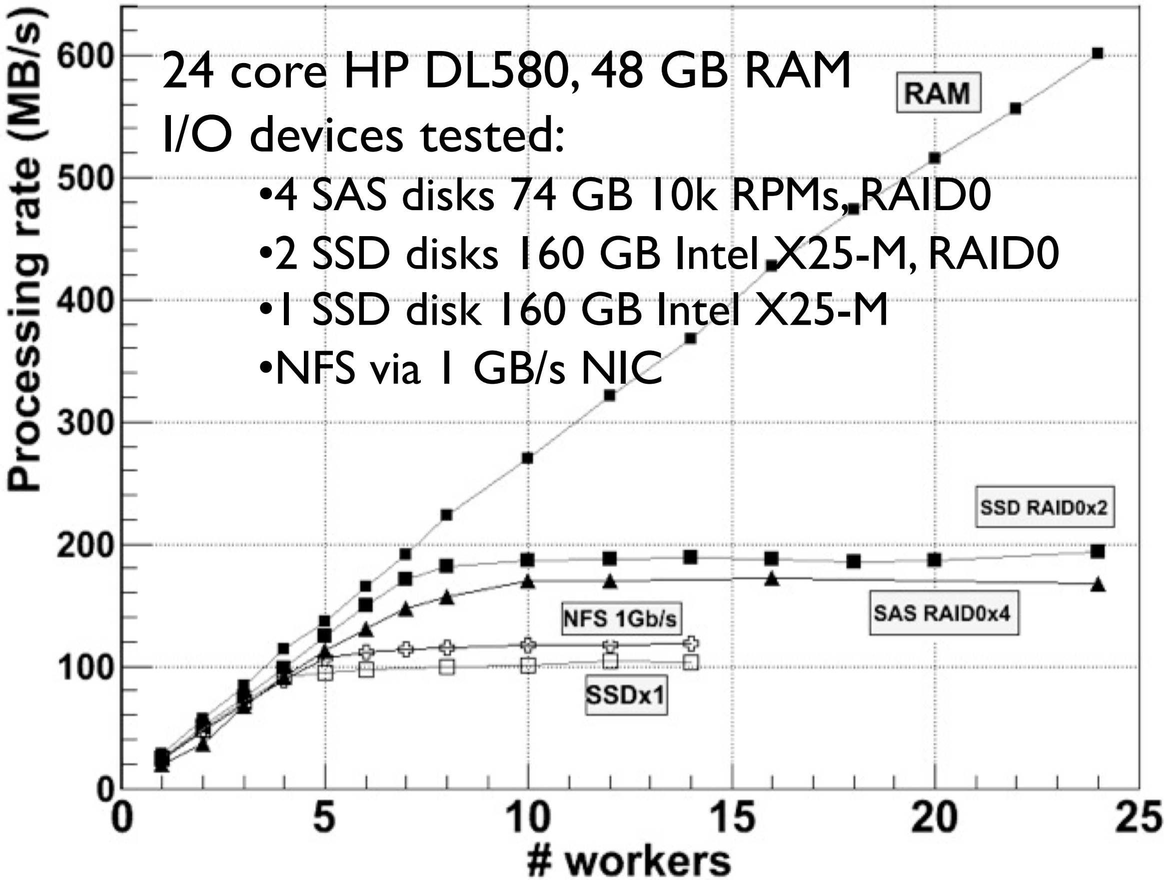


I/O device(s) on a single machine

24 core HP DL580, 48 GB RAM

I/O devices tested:

- •4 SAS disks 74 GB 10k RPMs, RAID0
- •2 SSD disks 160 GB Intel X25-M, RAID0
- •1 SSD disk 160 GB Intel X25-M
- •NFS via 1 GB/s NIC



Algorithm Parallelization

- Ultimate performance gain will come from parallelizing **algorithms** used in current LHC physics application software
  - » Prototypes using posix-thread, OpenMP and parallel gcclib
  - » On going effort in collaboration with OpenLab and Root teams to provide basic thread-safe/multi-thread library components
    - Random number generators
    - Parallel minimization/fitting algorithms
    - Parallel/Vector linear algebra
- Positive and interesting experience with MINUIT
  - » Parallelization of parameter-fitting opens the opportunity to enlarge the region of multidimensional space used in physics analysis to essentially the whole data sample.

RooFit/Minuit Parallelization

- RooFit implements the possibility to split the likelihood calculation over different threads
  - » Likelihood calculation is done on sub-samples
  - » Then the results are collected and summed
  - » You gain a lot using multi-cores architecture over large data samples, scaling almost with a factor proportional to the number of threads
- However, if you have a lot of free parameters, the bottleneck become the minimization procedure
  - » Split the derivative calculation over several MPI processes
  - » Possible to apply an hybrid parallelization of likelihood and minimization using a Cartesian topology (see A.L. CHEP09 proceeding, to be published on …)
    - Improve the scalability for case with large number of parameters and large samples
- Code already inside ROOT (since 5.26), based on Minuit2 (the OO version of Minuit)

Parallel MINUIT *Alfio Lazzaro and Lorenzo Moneta*

– Minimization of Maximum Likelihood or χ^2 requires iterative computation of the gradient of the NLL function

$$\left.\frac{\partial NLL}{\partial \hat{\theta}}\right|_{\hat{\theta}_0} \approx \frac{NLL(\hat{\theta}_0 + \hat{\mathrm{d}}) - NLL(\hat{\theta}_0 - \hat{\mathrm{d}})}{2\hat{\mathrm{d}}}$$

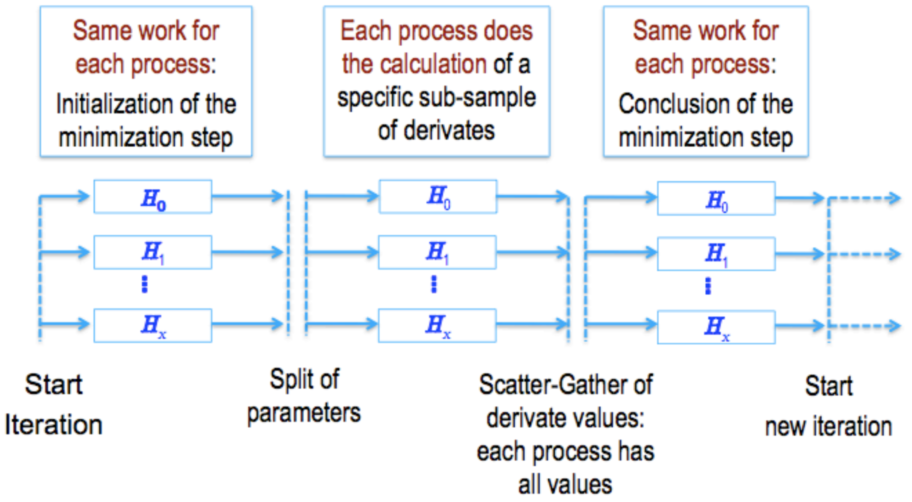
$$NLL = \ln\left(\sum_{j=1}^{s} n_j\right) - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j^i\right)$$

j species (signals, backgrounds)
n_j number of events for specie j
$\mathcal{P}_j$ probability density functions (PDFs)
N number total of events to fit

– Execution time scales with number θ free parameters and the number N of input events in the fit

– Two strategies for the parallelization of the gradient and NLL calculation:

1. Gradient **or** NLL calculation on the same multi-cores node (OpenMP)
1. Distribute Gradient on different nodes (MPI) **and** parallelize NLL calculation on each multi-cores node (pthreads): **hybrid solution**

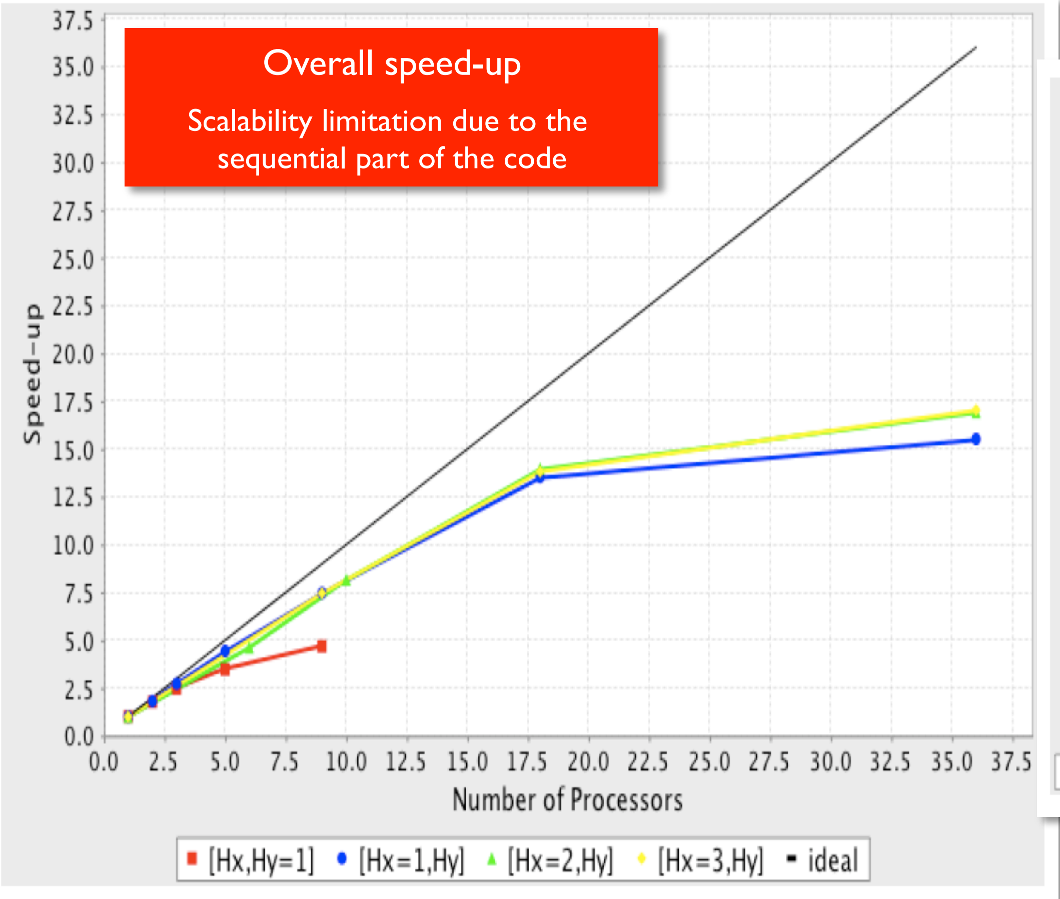


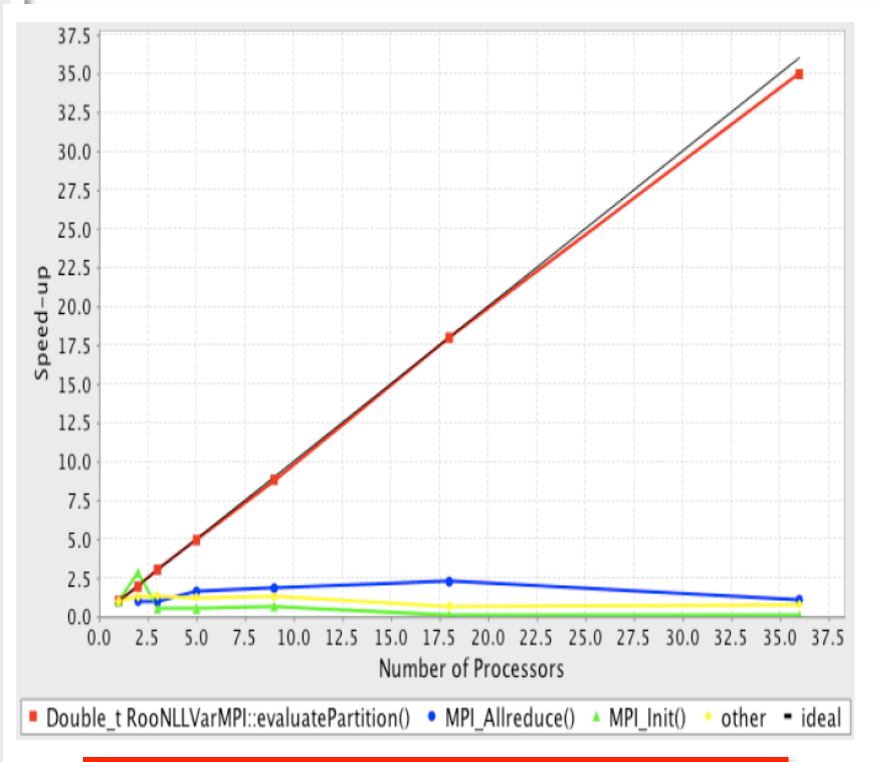
Test @ INFN CNAF cluster, Bologna (Italy)

3 variables, 600K events, 23 free parameters

PDFs per each variable: 2 Gaussians for signal, parabola for background

Sequential execution time (Intel Xeon @ 2.66GHz): **~80 minutes**





`RooNLLVarMPI::evaluatePartition()` does the NLL calculation: excellent scalability

DEPLOYMENT ISSUES

Need for Dedicated Batch Queues

LSF Testing : Results Using standard generic Queues

Parallel	nTests	Average Wait(s)	Max Wait (s)	Min Wait (s)
2	191	48.37	1072	3
3	170	412.29	8138	3
4	171	4608	35987	3
5	134	31345	137068	5
6	121	41990	136763	4

412s = 6m 52s
4608s = 1h 16m 48s
31345s = 8h 53m 25s
41990s = 11h 39m 50s

136763s = 37h 59m 23s

How to submit to OSG

```
universe = grid
GridResource = some_grid_host
GlobusRSL = MagicRSL
executable = wrapper.sh
arguments = arguments
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = inputs
transfer_output_files = output
queue
```

PBS
(host_xcount=1)(xcount=8)(queue=?)
LSF
(queue=?)(exclusive=1)
Condor
(condorsubmit=('+WholeMachine' true))

MPI and multi-thread support in EGEE : examples

PURE MULTI_THREAD

```
# e.g. single whole node with a minimum of 4 cores:
SMPGranularity = 4 ;
WholeNode = True ;
```

PURE MPI

```
#e.g. 16 MPI processes:
CPUnumber = 16 ;

# e.g. 16 MPI processes, whole nodes, a minimum of 4 cores each:
CPUnumber = 16 ;
SMPGranularity = 4 ;
WholeNode = True ;
```

HYBRID MULTI-THREAD/MPI

```
# e.g. 4 MPI processes, 1 per node, a minimum of 4 cores each:
NodeNumber = 4 ;
SMPGranularity = 4 ;
WholeNode = True ;
```

The Accounting Problem

By Matt Mackall

- We save memory by sharing it between processes
- ...but we count that memory multiple times when reporting it
- ...and we allocate more memory than is actually available
- The numbers don't add up!
- Users and developers can't get a good sense of how memory is used
- They end up bailing out the system by throwing more memory at it

http://www.selenic.com/smem/

Pagemap and friends

Matt Mackall

- In 2007, I attacked this problem from the kernel side with pagemap
- The pagemap interface exposes the mapping from virtual to physical memory and other details
- Along the way, two new concepts:
- PSS (Proportional Set Size)
  a mapping's fair share of shared memory
- USS (Unique Set Size)
  a mapping's non-overlapping memory usage
- ...and some proof-of-concept graphical tools

http://www.selenic.com/smem/

Memory accounting using **smem**:

15 cms reco processes forked by one master:

pretended total virtual memory used: 21GB, real: 5.7GB

smem

PID	User	Comm	Swap	USS	**PSS**	RSS
32116	innocent	top	0	616	651	1204
31962	innocent	-tcsh	0	1552	1789	2532
30747	innocent	-tcsh	0	2860	3309	3864
32123	innocent	/usr/bin/python /afs/cern.c	0	7216	7257	7880
31911	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	84176	137545	940336
31945	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	303436	357363	1170280
31936	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	304552	358555	1172184
31937	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	309060	362986	1175968
31944	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	309860	363762	1176520
31931	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	311472	365484	1179052
31939	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	313060	366972	1179796
31942	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	313232	367179	1180212
31943	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	313920	367814	1180312
31938	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	314840	368784	1181944
31935	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	315172	369093	1182048
31934	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	315220	369173	1182436
31933	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	315520	369491	1182824
31932	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	316208	370235	1183892
31940	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	318144	372083	1185212
31941	innocent	cmsRun reco_RAW2DIGI_RECO_p	0	329432	383356	1196240
70	1		0	4799548	**5662670**	18664736

top:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	P	CODE	DATA	COMMAND
31931	innocent	20	0	1315m	1.1g	133m	R	100.0	4.8	3:27.43	0	108	1.1g	cmsRun

Memory accounting using "smaps"

Developed in SFT by Pere Mato and Eoin Smith

/afs/cern.ch/sw/lcg/external/smaps/1.0

```
Process Summary at : Mon Mar  1 12:25:51 2010
--------------------------------------------------------------------------------------------------
-tcsh            29384 - Rss :   3592 - Size :   68452 - Code(priv/shar) :  0 / 872 - Data(priv/shar) :  2604 /   116
-tcsh            29800 - Rss :   3752 - Size :   68588 - Code(priv/shar) :  4 / 896 - Data(priv/shar) :  2732 /   120
cmsRun reco_R children=16  Rss :  940144 Size : 1075128 - Code(priv/shar) : 48/1256 - Data(priv/shar) :  84272/854568
cmsRun reco_R children=16  Rss : 1175932 Size : 1334852 - Code(priv/shar) : 0/1060 - Data(priv/shar)  308984/865888
cmsRun reco_R children=16  Rss : 1167384 Size : 1325148 - Code(priv/shar) : 0/1060 - Data(priv/shar) :  300404/865920
cmsRun reco_R children=16  Rss : 1178768 Size : 1337580 - Code(priv/shar) : 0/ 1060 - Data(priv/shar) : 311996/865712
cmsRun reco_R children=16  Rss : 1171224 Size : 1331516 - Code(priv/shar) : 0/1060 - Data(priv/shar) :  304596/865568
cmsRun reco_R children=16  Rss : 1182340 Size : 1337080 - Code(priv/shar) : 0/1060 - Data(priv/shar)  316080/865200
cmsRun reco_R children=16  Rss : 1170712 Size : 1327936 - Code(priv/shar) : 0/1060 - Data(priv/shar)  303708/865944
cmsRun reco_R children=16  Rss : 1174796 Size : 1330972 - Code(priv/shar) : 0/1060 - Data(priv/shar)  308208/865528
cmsRun reco_R children=16  Rss : 1180608 Size : 1336912 - Code(priv/shar) : 0/1060 - Data(priv/shar)  314188/865360
cmsRun reco_R children=16  Rss : 1179804 Size : 1337376 - Code(priv/shar) : 0/1060 - Data(priv/shar)  313760/864984
cmsRun reco_R children=16  Rss : 1185048 Size : 1343144 - Code(priv/shar) : 0/1060 - Data(priv/shar)  318624/865364
cmsRun reco_R children=16  Rss : 1185840 Size : 1346956 - Code(priv/shar) : 0/1060 - Data(priv/shar)  319400/865380
cmsRun reco_R children=16  Rss : 1180312 Size : 1340232 - Code(priv/shar) : 0/1060 - Data(priv/shar)  313892/865360
cmsRun reco_R children=16  Rss : 1177604 Size : 1337220 - Code(priv/shar) : 0/1060 - Data(priv/shar)  311888/864656
cmsRun reco_R children=16  Rss : 1175464 Size : 1334584 - Code(priv/shar) : 0/1060 - Data(priv/shar)  309460/864944
cmsRun reco_R children=16  Rss : 1150596 Size : 1310248 - Code(priv/shar) : 0/1060 - Data(priv/shar)  284504/865032
cmsRun reco_R children=16  Rss : 1184504 Size : 1343256 - Code(priv/shar) : 0/1060 - Data(priv/shar)  318240/865204
--------------------------------------------------------------------------------------------------
Total Size : 22038.26 Mb
Total Rss  : 19305.10 Mb
```

Real: ~ 300 x 16 + 85 + 865 MB = 5.75 GB

Summary

- The stagnant speed of single processors and the narrowing of the number of OSs and computing architectures modify the strategy to improve the performance of software applications
  - » Aggressive software optimization tailored to the processor in hand
  - » Parallelization
  - » Optimization of the use of “out-core” resources
- Experimental HEP is blessed by the natural parallelism of event processing:
  - » Very successful evolution of “frameworks” to multi-process with read-only shared memory
  - » Exploiting this new processing model requires a new model in computing resources allocation as well:
    - The most promising solution is full node allocation