HPC Programming Models, Compilers, Performance Analysis
IBM Systems – Infrastructure Solutions

Ludovic Enault, IBM
Geoffrey Pascal, IBM
Agenda

- System architecture trend overview
- Programming models & Languages
- Compiler
- Performance Analysis Tools
System architecture trend overview
What’s happening?

- Industry shift to multi-cores/many-cores, accelerators
  - Intel Xeon+PHI+FPGA, IBM POWER+GPU+FPGA, ARM+GPU+FPGA
- Increasing
  - # Cores
  - Heterogeneity with Unified Memory
  - Memory complexity

2 architectures:

- Single core
- Homogeneous Multi-core
- Many-core Heterogeneous
- Hybrid

Programming Model Complexity

- standard memory model: unique memory address space
- Non-standard memory model: separate memory address spaces,

GPU/FPGA
Accelerated Accelerators

Kepler
CUDA 5.5 – 7.0
Unified Memory

Pascal
CUDA 8
Full GPU Paging

Volta
CUDA 9
Cache Coherent

Kepler
1.5TF
12GB @ 288GB/s

Pascal
SXM2
NVLink 1.0
>40+40 GB/s
On Demand Paging

Volta
SXM2
NVLink 2.0
>75+75 GB/s
Coherent

POWER8
PCle
Buffered Memory

POWER8+

POWER9

2014-2015
2016
2017
Memory Hierarchy and data locality – single node

- Memory hierarchy tries to exploit locality
- CPU: low latency design
- Data transfers to accelerator are very costly
- Accelerator: high latency and high bandwidth
- Parallel Computing: architecture overview
Main architecture trends and characteristics

- More and more cores (CPU and GPU) per node with Simultaneous Multiple Threading (up to 8 on IBM Power)

- Accelerator integration with unified memory hierarchic
  \[\Rightarrow\text{performance requires data locality}\]

- Vector floating point units and SIMD (Single Instruction Multiple Data) operations
  \[\Rightarrow\text{Performance requires application vectorization (both operations and data)}\]

- Multiple levels of parallelism
Parallel Computing: architecture overview

Memory BUS
Memory

Uniform Memory access (UMA):
Each processor/processes has uniform access to memory
Shared Memory programming model

Cache Coherent Uniform Memory access (ccNUMA):
Time for memory access depends on data location. Local access is faster
Shared Memory programming model

Heterogeneous/Hybrid accelerated processor
Each accelerator has its own local memory and address space (changing)
Hybrid programming model

Accelerator
High performance local memory
GPU, FPGA, MIC

Accelerator
High performance local memory
GPU, FPGA, MIC
HPC cluster

**Distributed Memory:**
Each node has its own local memory. Must do message passing to exchange data between nodes (most popular approach is MPI)

Cluster Architecture
Computation is performed in multiple places.

A place contains data that can be operated on remotely.

Data lives in the place it was created, for its lifetime.

A datum in one place may reference a datum in another place.

Data-structures (e.g. arrays) may be distributed across many places.

Places may have different computational properties.
Where Does Performance Come From?

- Computer Architecture
  - Instruction issue rate
    - Execution pipelining
    - Reservation stations
    - Branch prediction
    - Cache & memory management
  - Parallelism
    - Parallelism – number of operations per cycle per processor
    - Parallelism – number of threads per core
    - Parallelism – number of cores per processor (SMT)
    - Parallelism – number of processors per node
    - Parallelism – number of accelerator per node
    - Parallelism – number of nodes in a system
- Device Technology
  - Memory capacity and access time
  - Communications bandwidth and latency
  - Logic switching speed and device density

Not anymore distributed and shared memory paradigms

Node
Socket
Chip
Core
Thread
Register/SIMD
Multiple instruction pipelines

Need to optimize for all levels!
HPC Programming models & languages

- data parallelism
- task parallelism

**Multicore, Manycore, accelerator, large scale**

**Standard ??**
- C/C++/Fortran OpenMP
- Python, R for Mapreduce/Spark

**Shared & Distributed Memory**
- UPC, CAF, ARMCI/Global Arrays,
- CUDA, OpenCL, OpenACC, CILK, HMPP,
- StarSc, X10, Chapel, Fortress, Sisal, ...

**Distributed Memory**
- OPenMP, TBB, pTheads, MparReduce...

**Shared Memory**
- C++, ADA, perl, Tcl, XML...

**Single Memory**
- High level: Fortran, LISP, COBOL, C, ...

**Assembler**


- Vector Units: SIMD
Programming languages & Programming models
Different ways to program and Accelerate Applications

Libraries
- Easy to use
- Most Performance
- Is there an existing library that can do what I want?

Compiler Directives
- Easy to use
- Portable code
- Can I easily add directives to help the compiler?

Specific Programming Languages
- Less portable
- Optimal performance
- Is it performance critical?
Programming languages

- 2 main types **languages**
  - Compiled: **C**, **C++**, **Fortran**, ADA…
    - Compilers: GCC, CLANG/LLVM, IBM XL, INTEL, NVIDIA PGI, PathScale, Visual C/C++
  - Interpreted: Python, java, R, Ruby, perl,…

- Many programming **models**
  - **Shared memory**
    - Pthreads APIs, **OpenMP/OpenACC** directives for C/C++/Fortran, TBB-Thread Building Blocks, CILK - Lightweight threads embedded into C, java threads, …
  - **Accelerator**
    - OpenMP4.x, OpenACC directives for C/C++/Fortran, CUDA& OpenCL APIs, libspe, ATI,, StarPU (INRIA), SequenceL, VHDL for FPGA, …
  - **Distributed memory**
    - MPI, Sockets, PGAS (UPC, CAF…), …

Strong focus and development effort for OpenMP (IBM, NVIDIA, INTEL)
High Performance Programming overview

- For a programmer language should not be the barrier. The critical points are
  - To identify and extract parallelism
  - the programming model, as from language to language mainly syntax changes

```
THINK PARALLEL
```

```
Identify Parallelism
Express Parallelism
Express Data Locality
Optimize
```

**critical**
Before choosing a programming model & languages

1. What parallelism you could extract?
2. What are the characteristics of your application?
3. Which curve are you on?
4. What are the current performances?
5. What performance do you need?
6. When do you want to reach your target?
7. What’s the life span of your application, versus hardware life span?
8. What are your technical resources and skills?
Programming models for HPC

- The challenge is to efficiently map a problem to the architecture
  - Address parallel paradigms for large futures systems (vector, threading, data-parallel and transfers, message-passing, accelerator...)
  - Address scalability
  - Take advantage of all computational resources
  - Support well performance programming
  - Take advantage of advances in compiler
  - Interoperable with existing languages
  - Guaranty portability

- For a programmer language should not be the barrier. The critical point is the programming model supported, other criteria: portability, simplicity, efficiency, readability

- Main languages for traditional HPC applications:
  - C/C++, Fortran, Python, R

- Languages evolution: more parallelism and hybrid computing feature (C++17, OpenMP 4.5, OpenACC 3.0, UPC, CAF ...)

Beyond multi-core and parallelism

- The problem is not multi-node, multi-core, many-core, ...

But

- The problem is in the application programmer’s head
  - Do I have parallelism?
  - What is the right programming model for concurrency or/and heterogeneity, efficiency, readability, manageability, ...?
  - Address clusters, SMPs, multi-cores, accelerators...

- Common trends
  - more and more processes and threads
  - Data centric

- How to estimate the development cost and impacts for
  - entrance
  - exit
Vectorization overview

each current and future core has vector units
SIMD – Single Instruction Multiple Data

- **Scalar Processing**
  - Traditional mode
  - One operation produces one result

- **SIMD Processing**
  - One operation produces multiple results

- parallel vector operations
- applies the same operation in parallel on a number of data items packed into a 128-512-bit vector (2-8 DP operation per cycle)
  - Without vector operation peak performance must divide by vector length
- There are many different versions of SIMD extensions
  - SSE, AVX, AVX2, AVX-512, Altivec, VMX
Vectorization example - Single DAXPY : A*X Plus Y

- 3 ways to enable vector operations: compiler, library and Intrinsic APIs

Using the compiler (« portable »)

```c
#define N 1000
void saxpy(float alpha, float *X, float *Y) {
    for (int i=0; i<N; i++)
        Y[i] = alpha*X[i] + Y[i];
}
```

- Aliasing prevents the compiler from doing vectorization
  - pointers to vector data should be declared with the restrict keyword
  - restrict means that we promise that there are no aliases to these pointers
- There is also an issue with access to unaligned data
  - the compiler can not know whether the pointers are aligned to 16 bytes or no

```c
#define N 1000
void saxpy(float alpha, float __restrict *X, float __restrict *Y) {
    for (int i=0; i<N; i++)
        Y[i] = alpha*X[i] + Y[i];
}
```

```c
#define N 1000
void saxpy(float alpha, float __restrict *X, float __restrict *Y) {
    float *a = __builtin_assume_aligned(X, 16);
    float *b = __builtin_assume_aligned(Y, 16)
    for (int i=0; i<N; i++)
        Y[i] = alpha*X[i] + Y[i];
}
```
Vectorization example - Single DAXPY: A*X Plus Y

- 3 ways to enable vector operations: compiler, library and Intrinsics functions

Using intrinsicsic (« not portable »)

Example 128-bit MMX – prefix _mm_

process: declare vectors, load/store vector, vector operations

```c
#include <emmintrin.h>
#define N 1000
void saxpy(float alpha, float *X, float *Y) {
  __m128 x_vec, y_vec, a_vec, res_vec;
  a_vec = _mm_set1_ps(alpha);
  for (int i=0; i<N; i+=4) {
    x_vec = _mm_loadu_ps(&X[i]);
    y_vec = _mm_loadu_ps(&Y[i]);
    res_vec = _mm_add_ps(_mm_mul_ps(a_vec, x_vec), y_vec);
    _mm_storeu_ps(&Y[i], res_vec);
  }
}
```

Programming Models and Languages examples

– Shared Memory with OpenMP

– Distributed Memory with MPI, UPC, CAF, MapReduce/Spark
OpenMP compiler directive syntax

- **OpenMP**
  - **C/C++**
    
    ```c
    #pragma omp target directive [clause [,] clause]...
    ```
    
    ...often followed by a structured code block
  - **Fortran**
    
    ```fortran
    !$omp target directive [clause [,] clause] ...
    ```
    
    ...often paired with a matching end directive surrounding a structured code block:

    ```fortran
    !$omp end target directive
    ```
OpenMP: work distribution

\begin{verbatim}
!$OMP PARALLEL DO
    do i=1,32
        a(i)=a(i)*2
    end do
\end{verbatim}
OpenMP memory domain

- Multiple threads share global memory
- Most common variant: OpenMP
- Program loop iterations distributed to threads, more recent task features
  - Each thread has a means to refer to private objects within a parallel context
- Terminology
  - Thread, thread team
- Implementation
  - Threads map to user threads running on one SMP node
  - Extensions to distributed memory not so successful
- OpenMP is a good model to use within a node
Distributed Memory Message Passing (MPI) model

Message Passing concept:
If a message is sent to a process, this process must receive it
Message Passing

- Participating processes communicate using a message-passing API
- Remote data can only be communicated (sent or received) via the API
- MPI (the Message Passing Interface) is the standard
- Implementation: MPI processes map to processes within one SMP node or across multiple networked nodes
- API provides process numbering, point-to-point and collective messaging operations
- Mostly used in two-sided way, each endpoint coordinates in sending and receiving
Map Reduce runtime

Example: count the # of occurrences of each work in large collection of documents

MapReduce runtime manages transparently the parallelism

map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += parseInt(v);
    Emit(AsString(result));

Map invocations are distributed across multiple machine by automatically partitioning the input data in M splits or shards.

reduce invocations are distributed by partitioning the intermediate key space into R pieces

# partitions are specified by user
MapReduce processing scheme
- Simple example OpenMP and OpenACC for both CPU and GPU
  - Express parallelism and manage data locality
OpenMP and OpenACC in Fortran/C/C++ for parallel computing

- Compiler directives advantages
  - shared and hybrid parallelization
    - Work and task parallelization
    - Data control location and movement
  - portable
  - processor and acceleration support
  - code changes limitation
  - Committed to pre-exascale architectures
OpenMP and OpenACC Directive syntax

• **OpenMP**
  • **C/C++**
    
    #pragma omp target directive [clause [,] clause]...
    
    ...often followed by a structured code block
  • **Fortran**
    
    !$omp target directive [clause [,] clause] ...]
    
    ...often paired with a matching end directive surrounding a structured code block:
    
    !$omp end target directive

• **OpenACC**
  • **C/C++**
    
    #pragma acc directive [clause [,] clause]...
    
    ...often followed by a structured code block
  • **Fortran**
    
    !$acc directive [clause [,] clause] ...]
    
    ...often paired with a matching end directive surrounding a structured code block:
    
    !$acc directive
SAXPY in C

```c
void saxpy(int n, float a,
           float *x, float *y)
{
    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];

    int N = 1<<20;

    // Perform SAXPY on 1M elements
    saxpy(N, 2.0, x, y);
}
```

SAXPY in Fortran

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(*), y(*), a
    integer :: n, i
    !$omp parallel do
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$omp end parallel do
end subroutine saxpy

...!

! Perform SAXPY on N elements
call saxpy(N, 2.0, x, y)
...
SAXPY – Single prec A*X Plus Y in OpenACC - CPU&Accelerator

**SAXPY in C**

```c
void saxpy(int n, float a, float *x, float *y)
{
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(*), y(*), a
    integer :: n, i
    !$acc parallel loop
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$acc end parallel
end subroutine saxpy

... ! Perform SAXPY on N elements
saxpy(N, 2.0, x, y)
```
SAXPY – Single prec A*X Plus Y in OpenMP – Accelerator (GPU)

**SAXPY in C**

```c
void saxpy(int n, float a, 
    float *x, float *y)
{
    #pragma omp target teams \ 
    distribute parallel for
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];

    int N = 1<<20;

    // Perform SAXPY on 1M elements
    saxpy(N, 2.0, x, y);
}
```

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(*), y(*), a
    integer :: n, i
    !omp target teams & 
    !omp distribute parallel do
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !omp end target teams & 
    !omp distribute parallel do
end subroutine saxpy

... 
! Perform SAXPY on N elements 
call saxpy(N, 2.0, x, y)
... 
```
Single example about how to express parallelism and data locality using compiler directives languages using a GPU accelerator.

Identify Parallelism  Express Parallelism  Express Data Locality  Optimize

Data must be transferred between CPU and GPU memories.

1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC
Example: Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
  - Common, useful algorithm
  - Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$

$$A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}$$
while ( err > tol && iter < iter_max ) {
    err=0.0;

    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
while ( err > tol && iter < iter_max ) {
    err=0.0;

    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma omp parallel for shared(m, n, Anew, A) reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for shared(m, n, Anew, A)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
Building the code

$ pgcc -acc -ta=nvidia:5.5,kepler -Minfo=accel -o laplace2d_acc laplace2d.c

main:

56, **Accelerator kernel generated**
57, #pragma acc loop gang /* blockIdx.x */
59, #pragma acc loop vector(256) /* threadIdx.x */
56, Generating present_or_copyout(Anew[1:4094][1:4094])
56, Generating present_or_copyin(A[0:][0:])
59, Generating NVIDIA code
59, Generating compute capability 3.0 binary
63, Loop is parallelizable
68, **Accelerator kernel generated**
69, #pragma acc loop gang /* blockIdx.x */
71, #pragma acc loop vector(256) /* threadIdx.x */
68, Generating present_or_copyin(Anew[1:4094][1:4094])
68, Generating present_or_copyout(A[1:4094][1:4094])
68, Generating NVIDIA code
68, Generating compute capability 3.0 binary
71, Loop is parallelizable
Why is OpenACC so much slower?
Why?
Profiling an OpenACC Application

$ nvprof ./laplace2d_acc  
Jacobi relaxation Calculation: 4096 x 4096 mesh
==10619== NVPROF is profiling process 10619, command: ./laplace2d_acc
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000403
  700, 0.000345
  800, 0.000302
  900, 0.000269
total: 134.259326 s
==10619== Profiling application: ./laplace2d_acc
==10619== Profiling result:
Time(%)  Time     Calls     Avg       Min       Max       Name
 49.59%  44.0095s  17000  2.5888ms  864ns  2.9822ms  [CUDA memcpy HtoD]
 45.06%  39.9921s  17000  2.3525ms  2.4960us  2.7687ms  [CUDA memcpyDtoH]
  2.95%   2.6162s  1000  2.6162ms  2.6044ms  2.6319ms  main_56_gpu
  2.39%   2.1188s  1000  2.1188ms  2.1023ms  2.1374ms  main_68_gpu
  0.01%  12.431ms  1000  12.430us  12.192us  12.736us  main_63_gpu_red
while ( err > tol && iter < iter_max )
{
    err=0.0;
    #pragma acc parallel loop reduction(max:err)
    for(int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
}

=> Need to use directive to control data location and transfers
# Jacobi Iteration: OpenACC C Code

```c
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc parallel loop
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

Copy A to/from the accelerator only when needed. Create Anew as a device temporary.
Speed-Up (Higher is Better)

- Single Thread
- 2 Threads
- 4 Threads
- 6 Threads
- OpenACC
- Programming Languages and Models vs Compilers
  - Accelerator or CPU performance?

- OpenACC:
  - PGI, (GCC 6.x 7.x), omnicompiler, ...

- OpenMP 4.x
  - CLANG, IBM XL compiler, Intel Compiler, ...
Performance Analysis Tools
2 types of performance monitoring

- System
  - % CPU utilization
  - Memory utilization
  - Disk
  - Network
  - Energy
  - Hardware counter

- Application
  - CPU Profiling per function/line/...
  - Communication Profiling
  - Hardware counters
  - Thread profiling
  - Hard counters
  - IO profiling
  - Memory

- Other counters
  - Application monitoring
  - System monitoring
System monitoring

- Some performance tools:
  - Linux
    - top, htop, nmon, netstat, lpcpu, iostat, sar, dstat, …
  - Framework
    - Ganglia
    - Collectd/graphit/grafana
    - …

- System data
  - CPU
  - Memory
  - Disks
  - Networks/ports
  - File Systems
  - process/threads
  - Locality/affinity/
  - …
  - …

- Report + Automated-intelligent assist
top / htop

- System monitoring
  - Core usage
  - Memory usage
  - Process information
    - Running status
    - Owner
- Monitor the node
  - Limited by operating system
Nmon (http://nmon.sourceforge.net/pmwiki.php)

- display CPU, GPU, energy, memory, network, disks (mini graphs or numbers), file systems, NFS, top processes, resources…
- Command `nmon`
Application performance analysis tools

- Sampling vs instrumented instrumentation
  - Sampling limited overhead
  - Instrumented requires filters to reduce overhead

- Main debugers
  - gdb, TotalView, allinea (DDT)

- Some performance tools
  - Linux
    - GNU CPU profiling, Perf, Valgrind, …
  - Framework
    - Intel Suite
    - Scalasca, TAU/paraprof/PerfExlorer, persiscope
    - Paraver
    - Allinea-MAP/Performance Reports
    - NVIDIA nvvp, OpenCL visual profiler
    - Vampir
    - JPrInterl suiteofiler …
  - …

- Performance data
  - MPI stats
  - OpenMP stats
  - Hardware counters & derived metrics
  - I/Os stats
  - CPU profile
  - Data transfer stats
  - Power consumption
  - …

- Automated-intelligent assist
Code profiling

- **Purpose**
  - Identify most-consuming routines of a binary
    - In order to determine where the optimization effort has to take place

- **Standard Features**
  - Construct a display of the functions within an application
  - Help users identify functions that are the most CPU-intensive
  - Charge execution time to source lines

- **Methods & Tools**
  - GNU Profiler, Visual profiler, addr2line linux command, …
    - new profilers mainly based on Binary File Descriptor library and **opcodes** library to assemble and disassemble machine instructions
    - Need to compiler with `-g`
  - Hardware counters

- **Notes**
  - Profiling can be used to profile both serial and parallel applications
  - Based on sampling (support from both compiler and kernel)
GNU Profiler (Gprof) | How-to | Collection

- Compile the program with options: `-g -pg`
  - Will create symbols required for debugging / profiling

- Execute the program
  - Standard way

- Execution generates profiling files in execution directory
  - `gmon.out.<MPI Rank>`
    - Binary files, not readable
  - Necessary to control number of files to reduce overhead

- Two options for output files interpretation
  - GNU Profiler (Command-line utility): `gprof`
    - `gprof <Binary> gmon.out.<MPI Rank> > gprof.out.<MPI Rank>`

- Advantages of profiler based on Binary File Descriptor versus gprof
  - Recompilation not necessary (linking only)
  - Performance overhead significantly lower
GNU profile overview

- Step 1: compile code with ‘-pg’ option:
  - `$ gcc -Wall -pg test_gprof.c test_gprof_new.c -o test_gprof`
  - `$ ls`
    - test_gprof test_gprof.c test_gprof_new.c

- Step 2: execute code
  - `./test_gprof`
  - `$ ls`
    - gmon.out test_gprof test_gprof.c test_gprof_new.c

- Step 3: run the gprof tool
  - `$ gprof test_gprof gmon.out > analysis.txt`
  - `$ cat analysis.txt`

```
Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>cum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>self</td>
<td></td>
</tr>
<tr>
<td>time seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>33.86</td>
<td>15.52</td>
<td>1</td>
</tr>
<tr>
<td>33.82</td>
<td>31.02</td>
<td>1</td>
</tr>
<tr>
<td>33.29</td>
<td>46.27</td>
<td>1</td>
</tr>
<tr>
<td>0.07</td>
<td>46.30</td>
<td>0.03</td>
</tr>
</tbody>
</table>
```

% the percentage of the total running time of the
time program used by this function.
Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple commandline interface.

perf

usage: perf [--version] [--help] COMMAND [ARGS]

The most commonly used perf commands are:
- annotate: Read perf.data (created by perf record) and display annotated code
- archive: Create archive with object files with build-ids found in perf.data file
- bench: General framework for benchmark suites
- buildid-cache: Manage <tt>build-id</tt> cache.
- buildid-list: List the buildids in a perf.data file
- diff: Read two perf.data files and display the differential profile
- inject: Filter to augment the events stream with additional information
- kmem: Tool to trace/measure kernel memory(slab) properties
- kvm: Tool to trace/measure kvm guest os
- list: List all symbolic event types
- lock: Analyze lock events
- probe: Define new dynamic tracepoints
- record: Run a command and record its profile into perf.data
- report: Read perf.data (created by perf record) and display the profile
- sched: Tool to trace/measure scheduler properties (latencies)
- script: Read perf.data (created by perf record) and display trace output
- stat: Run a command and gather performance counter statistics
- test: Runs sanity tests.
- timechart: Tool to visualize total system behavior during a workload
- top: System profiling tool.

See 'perf help COMMAND' for more information on a specific command.
perf Linux serial execution

```
perf stat -B dd if=/dev/zero of=/dev/null count=1000000
1000000+0 records in
1000000+0 records out
512000000 bytes (512 MB) copied, 0.956217 s, 535 MB/s

Performance counter stats for 'dd if=/dev/zero of=/dev/null count=1000000':

  5,099 cache-misses    # 0.005 M/sec (scaled from 66.58%)
 235,384 cache-references # 0.246 M/sec (scaled from 66.56%)
 9,281,660 branch-misses # 3.858 %  (scaled from 33.50%)
 240,609,766 branches    # 251.559 M/sec (scaled from 33.66%)
 1,403,561,257 instructions # 0.679 IPC (scaled from 50.23%)
 2,066,201,729 cycles    # 2160.227 M/sec (scaled from 66.67%)
  217 page-faults        # 0.000 M/sec
  3 CPU-migrations       # 0.000 M/sec
  83 context-switches    # 0.000 M/sec
 956.474238 task-clock-msecs # 0.999 CPUs

0.957617512    seconds time elapsed
```

```
perf stat -B -e cycles,cycles ./noploop 1

Performance counter stats for './noploop 1':

  2,812,305,464 cycles
  2,812,304,340 cycles

        1.302481065    seconds time elapsed
```
perf Linux MPI execution execution

- mpirun [mpirun_options] mpyperf.sf execution [args]
- cat myperf.sh
  ```bash
  #!/bin/bash
  driver=
  if [ $PMI_RANK -eq 0 ]; then
    driver="perf record -e cycles -e instructions -o perf.data.$PMI_RANK"
  fi
  $driver "$@
  ```

(***) Check our mpi library and batch scheduler to get MPI rank variable
Valgrind

- Memory checker and profiler
- Not interactive
- Add overhead during execution
- Have to integrate symbols in your code (compile with flag ‘-g’ with Intel Compiler and GCC)
- Give information about:
  - Memory overflow
  - Undefined variable
  - Unallocated memory at the end of the execution
  - Double free corruption (release an already freed memory)

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>valgrind &lt;program&gt;</td>
<td>Perform regular memory checking</td>
</tr>
<tr>
<td>Valgrind --v &lt;program&gt;</td>
<td>Verbose mode</td>
</tr>
<tr>
<td>valgrind --leak-check=full &lt;program&gt;</td>
<td>Perform memory leak checking</td>
</tr>
</tbody>
</table>
INTEL MPI Profiling: STAT

Use lightweight statistics

- Set I_MPI_STATS to a non-zero integer value to gather MPI communication statistics (max value is 10)
- Manipulate the results with I_MPI_STATS_SCOPE to increase effectiveness of the analysis
- Example on the right - Gromacs rank 0 with suggested values
- Suggested values:

```
$ export I_MPI_STATS=3; export I_MPI_STATS_SCOPE=coll
```
INTEL MPI Profiling: ITAC (vampire, TAU, ...)

Start with simple default MPI only trace: `mpirun -trace ...`

Full instrumentation using `-tcollect`

Simple MPI Trace – Trace File Size

All computation is named Application

Many calls to `MPI_COMM_SIZE` increase file size
INTEL MPI Profiling: MPS

MPI Performance Snapshot

Delivered with Intel® Trace Analyzer & Collector (ITAC)
- Separated tools for statistical analysis and event analysis
- Available now via command line and optional html summary page

New capabilities available to developers
- MPS enables the developer to quickly gather and analyze statistics on up to 37,000 ranks (tested)
- Shows PAPI or Perf counters and MPI- & OpenMP imbalances
- Enables Intel Trace Analyzer and Collector trace file targeted for deeper event based analysis
Scalasca (http://www.scalasca.org/) – open source

- Scalasca is a software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes.

- Reference preparation for validation
  0. Program instrumentation: skin
  1. Summary measurement collection & analysis: scan [-s]
  2. Summary analysis report examination: square
  3. Summary experiment scoring: square -s
  4. Event trace collection & analysis: scan -t
  5. Event trace analysis report examination: square
Scalasca analysis report exploration 1/2

Distribution of selected metric for call path by process/thread
Scalasca analysis report exploration 2/2

Same message statistics, but times greatly reduced
TAU

- TAU = Tuning and Analysis Utility
  - Program and performance analysis tool framework being developed for the DOE Office of Science, ASC initiatives at LLNL, the ZeptoOS project at ANL, and the Los Alamos National Laboratory
  - Provides a suite of static and dynamic tools that provide graphical user interaction and interoperation to form an integrated analysis environment for parallel Fortran, C++, C, Java, and Python applications
  - Link
    - http://www.cs.uoregon.edu/research/tau/home.php