HPC Programming Models, Compilers, Performance Analysis

IBM Systems – Infrastructure Solutions

Ludovic Enault, IBM
Geoffrey Pascal, IBM

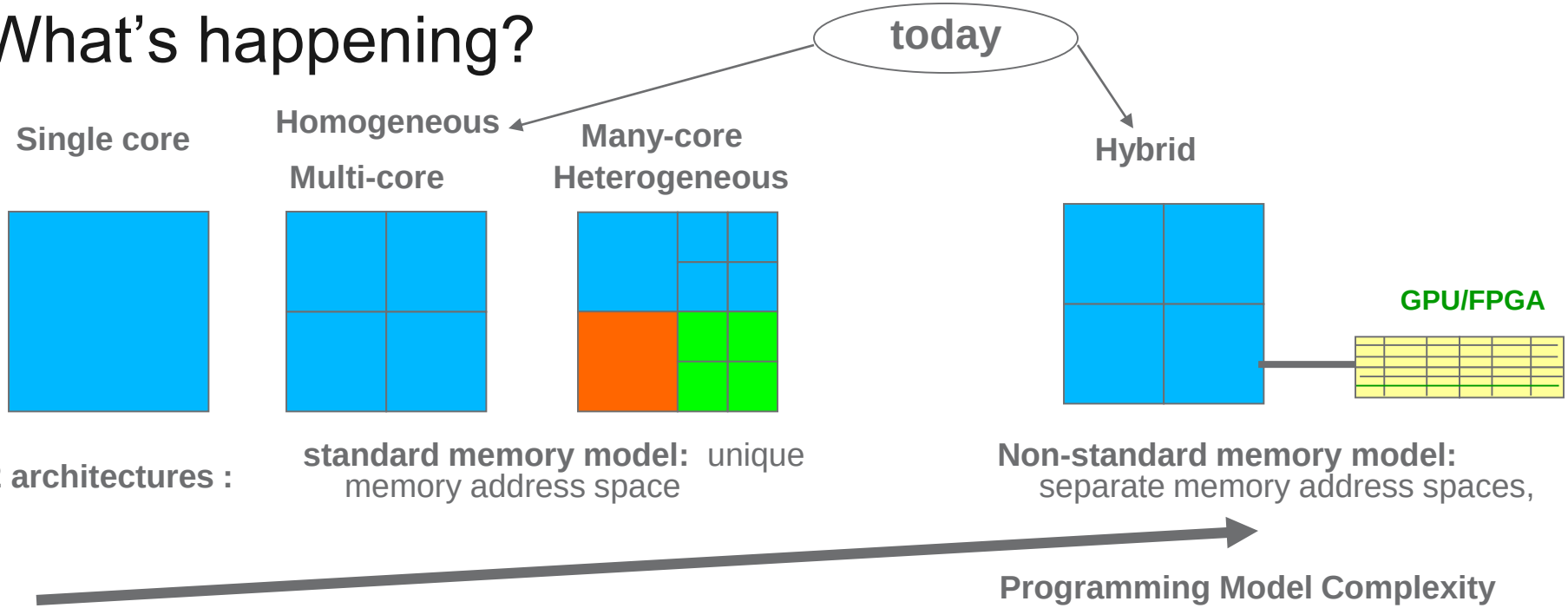
IBM **Client Center**

Agenda

- System architecture trend overview
- Programming models & Languages
- Compiler
- Performance Analysis Tools

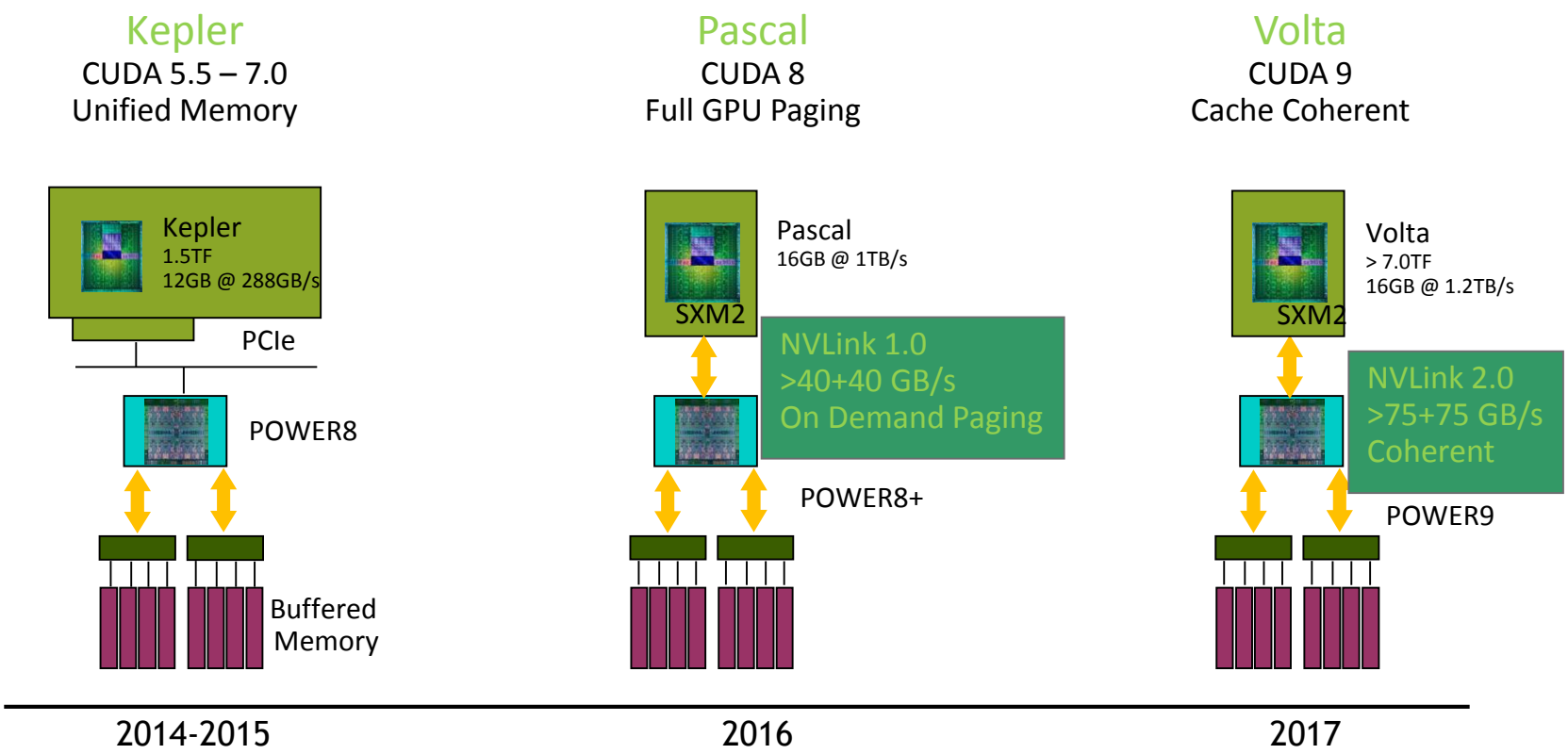
System architecture trend overview

What's happening?

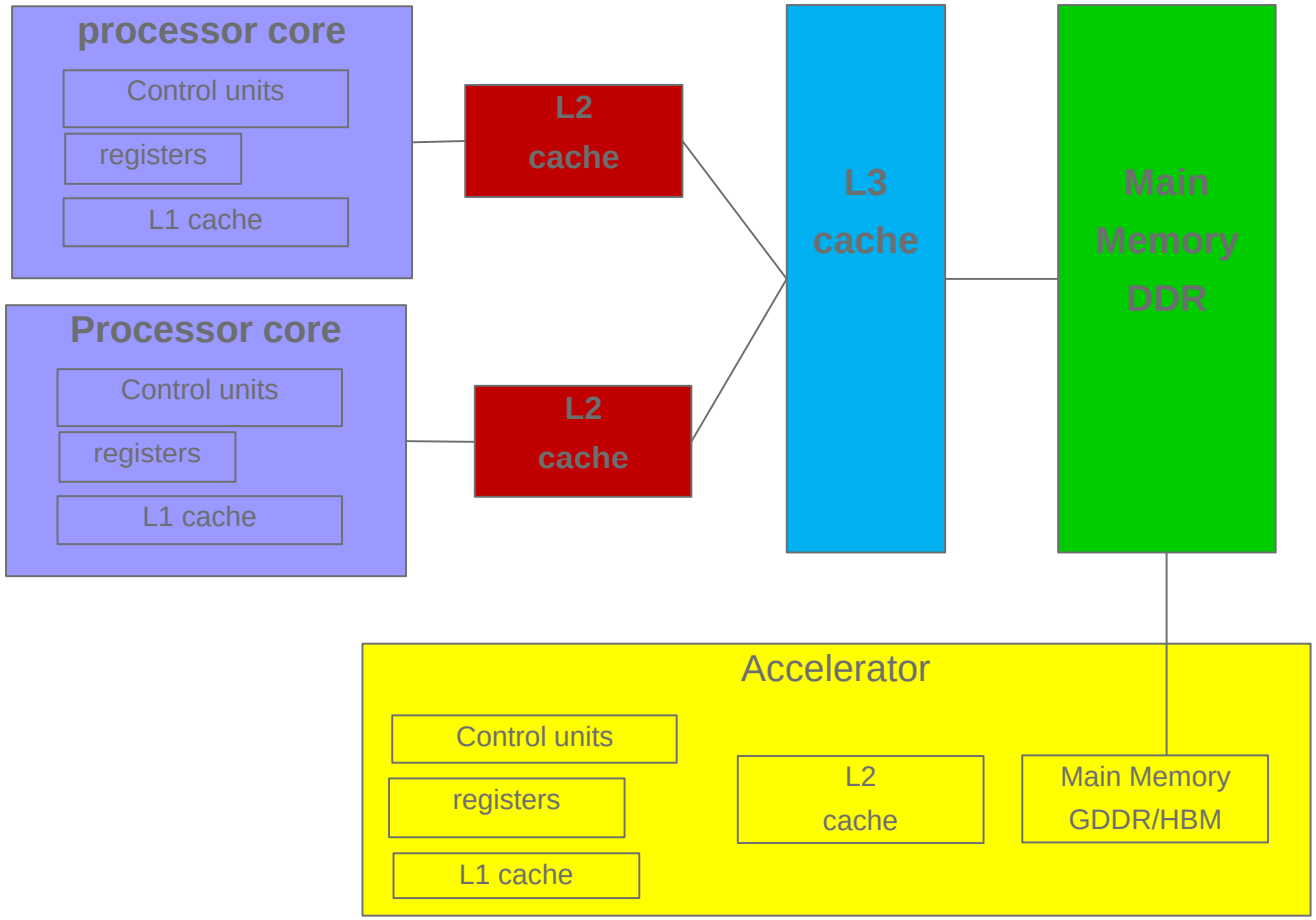


- Industry shift to multi-cores/many-cores, accelerators
  - Intel Xeon+PHI+FPGA, IBM POWER+GPU+FPGA, ARM+GPU+FPGA
- Increasing
  - # Cores
  - Heterogeneity with Unified Memory
  - Memory complexity

Accelerated Accelerators



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

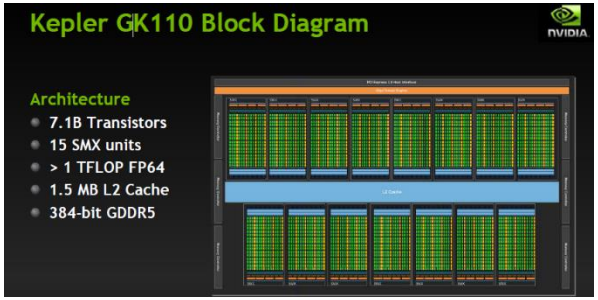
Main architecture trends and characteristics

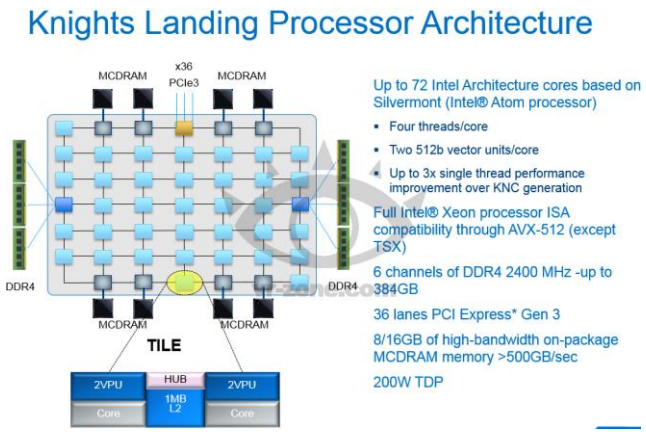
- More and more cores (CPU and GPU) per node with Simultaneous Multiple Threading (up to 8 on IBM Power)

IBM P8 cores 12 cores
4 GHz
128-bit FPU

Intel Broadwell
8-18 Cores ~3GHz
256-bit FPU

AMD
16-24 core/MCM
256-bit FPU



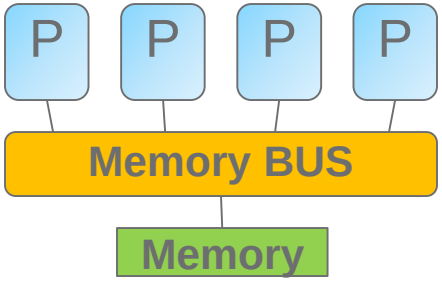


- Accelerator integration with unified memory hierarchic
  $\Rightarrow$ performance requires **data locality**

- Vector floating point units and SIMD (Single Instruction Multiple Data) operations
  => Performance requires application **vectorization** (both operations and data)

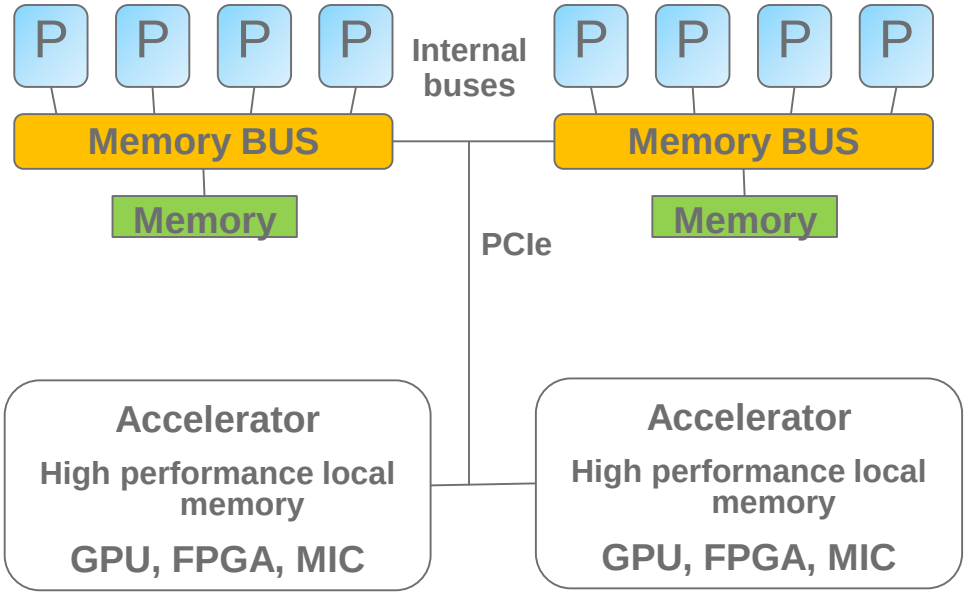
- Multiple levels of parallelism

Parallel Computing: architecture overview



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

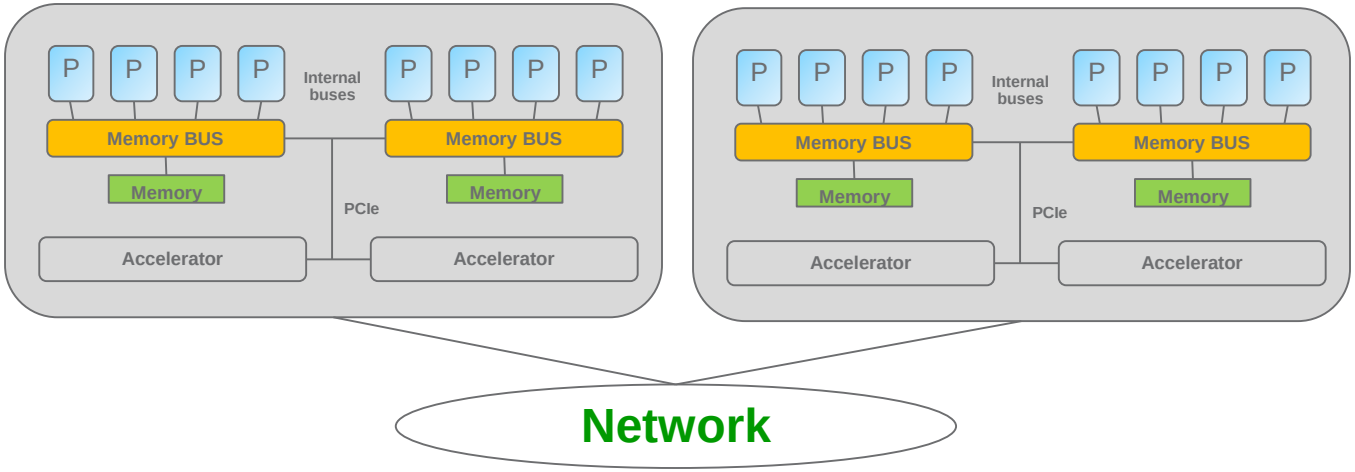
Cache Coherent Uniform Memory access (ccNUMA):
Time for memory access depends on data location. Local access is faster
Shared Memory programming model



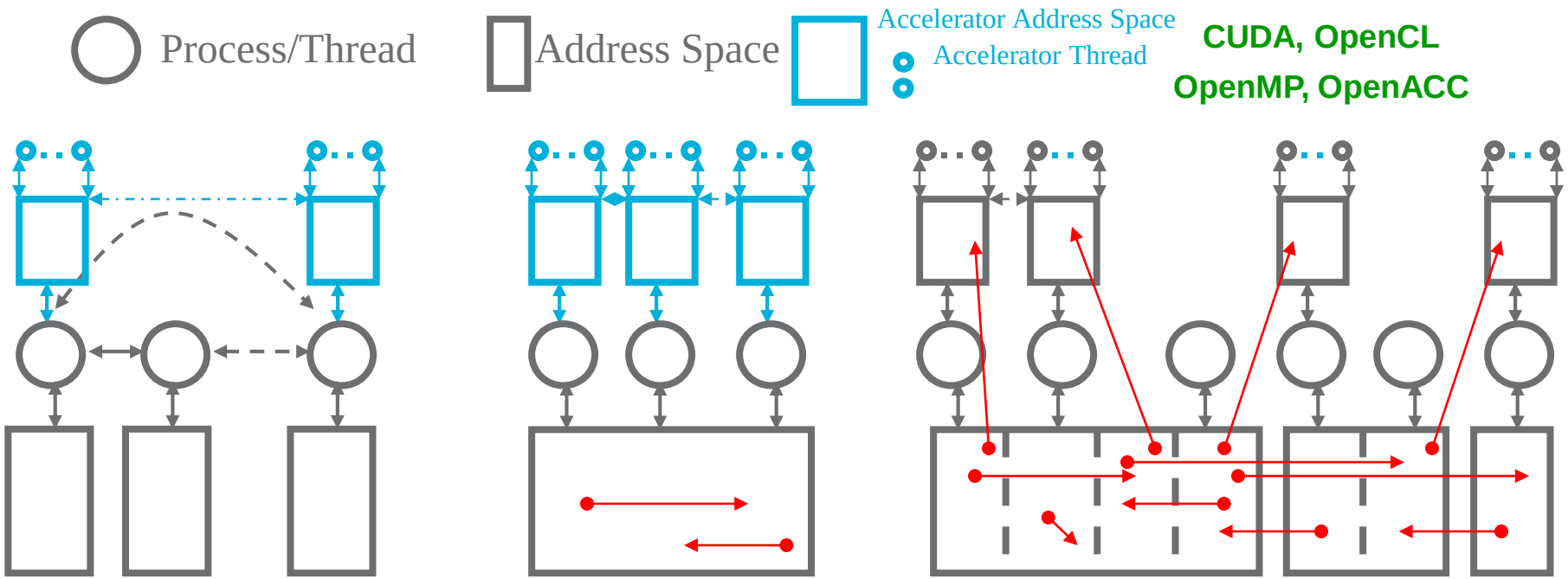
Heterogeneous/Hybrid accelerated processor
Each accelerator has it's own local memory and address space (changing)
Hybrid programing model

HPC cluster

Distributed Memory :
Each node has it's own local memory. Must do message passing to exchange data between nodes (most popular approach is MPI)
Cluster Architecture



Programming Models and Languages



Message passing

MPI

Shared Memory

pThreads, OpenMP, OpenACC,Java

PGAS

UPC, CAF, X10

- 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?

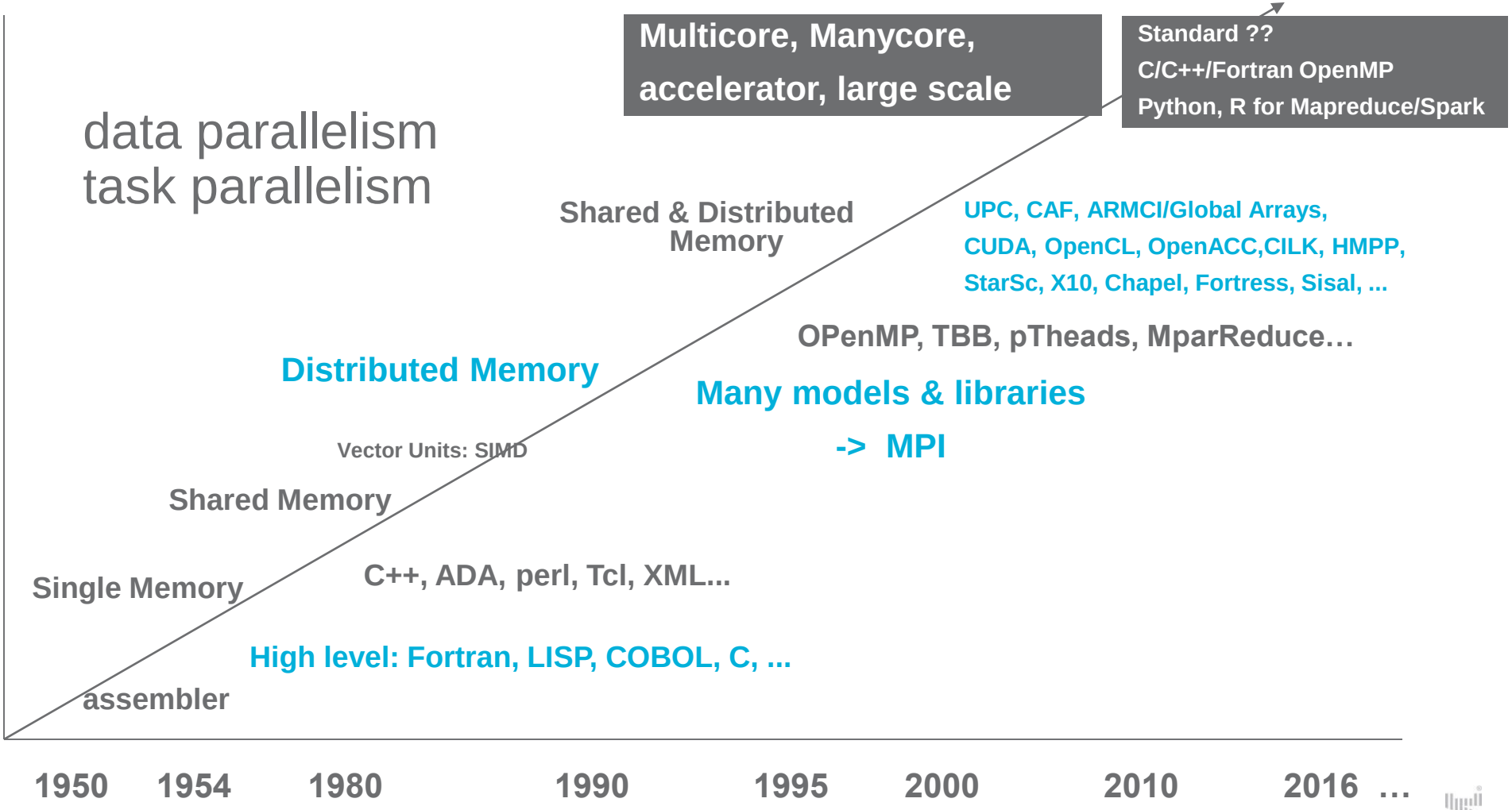
- Computer Architecture
  - Instruction issue rate
    - Execution pipelining
    - Reservation stations
    - Branch prediction
    - Cache & memory management
  - **Parallelism**
    - **Parallelism – number of operations per cycle per processor**
    - **Parallelism – number of threads per core**
    - **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



Programming languages & Programming models

Different ways to program and Accelerate Applications

Applications

Libraries	Compiler Directives OpenMP/OpnenACC/…	Specific Programming Languages
Easy to use **Most Performance**	**Easy to use** **Portable code**	**Less portable** **Optimal performance**
Is there an existing library that can do what I want ?	**Can I easily add directives to help the compiler ?**	**Is it performance critical ?**

Programming languages

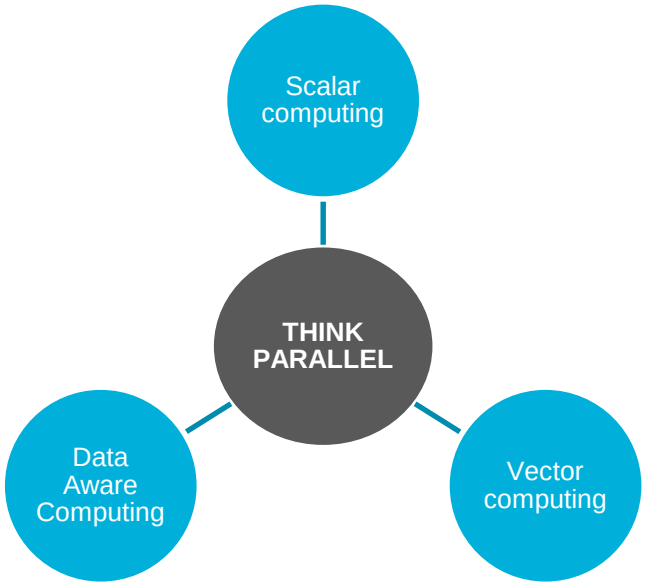
- 2 main types languages
  - Compiled: **C**, **C++**, **Fortran**, ADA…
    - Compilers: GCC, CLANG/LLVM, IBM XL, INTEL, NVIDIA PGI, PathScale, Visual C/C++
  - Interpreted: Python, java, R, Ruby, perl,…

- 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)

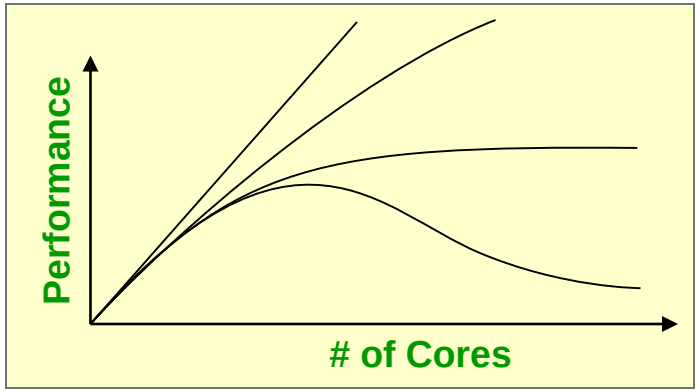
High Performance Programming overview

- 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



Before chosing a programming model & languages

1. What paralellism 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**

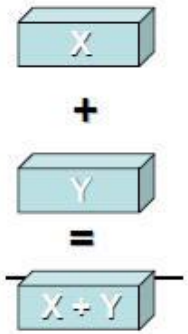
- How to estimate the development cost and impacts for
  - **entrance**
  - **exit**

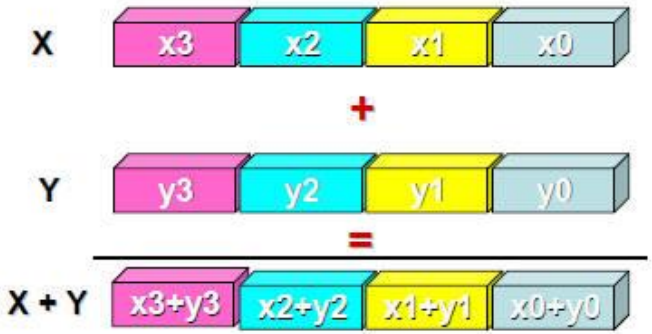
Vectorization overview

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 lenght
- 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 Instrinsic APIs

Using the compiler (« portable »)

```
#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];
}
```

```
gcc -O3 -c -std=c99 -fopt-info-optimized saxpy.c
saxpy.c:8:3: note: loop vectorized
saxpy.c:8:3: note: loop versioned for vectorization because of
                   possible aliasing
saxpy.c:8:3: note: loop peeled for vectorization to enhance alignment
saxpy.c:8:3: note: loop with 3 iterations completely unrolled
saxpy.c:7:6: note: loop with 3 iterations completely unroll
```

- Aliasing prevents the compiler from doing vectorization
  - pointers to vector data should be declared with the restric 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

```
#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];
}
```

```
gcc -O3 -c -std=c99 -fopt-info-optimized saxpy1.c
saxpy1.c:10:3: note: loop vectorized
saxpy1.c:10:3: note: loop peeled for vectorization to enhance
alignment
saxpy1.c:10:3: note: loop with 3 iterations completely unrolled
saxpy1.c:7:6: note: loop with 3 iterations completely unrolled
```

```
#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];
}
```

```
gcc -O3 -c -std=c99 -fopt-info-optimized saxpy2.c
saxpy2.c:10:3: note: loop vectorized
```

Vectorization example - Single DAXPY : A*X Plus Y

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

Using intrinsitic (« not portable »)

Example 128-bit MMX – prefix _mm_

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

```
#include <emmintrin.h>
#define N 1000
void saxpy(float alpha, float *X, float *Y) {
  __m128 x_vec, y_vec, a_vec, res_vec;          /* Declare vector variables */
  a_vec = _mm_set1_ps(alpha);                   /* Vector of 4 alpha values */
  for (int i=0; i<N; i+=4) {
       x_vec = _mm_loadu_ps(&X[i]);             /* Load 4 values from X */
       y_vec = _mm_loadu_ps(&Y[i]);             /* Load 4 values from Y */
      res_vec = _mm_add_ps(_mm_mul_ps(a_vec, x_vec), y_vec);   /* Compute */
      _mm_storeu_ps(&Y[i], res_vec);            /* Store the result */
    }
}
```

Programming Models and Languages examples

- Shared Memory with OpenMP
- Distributed Memory with MPI, UPC, CAF, MapReduce/Spark

OpenMP compiler directive syntax

- OpenMP
  - C/C++

    `#pragma omp target` *`directive [clause [,] clause]…]`*

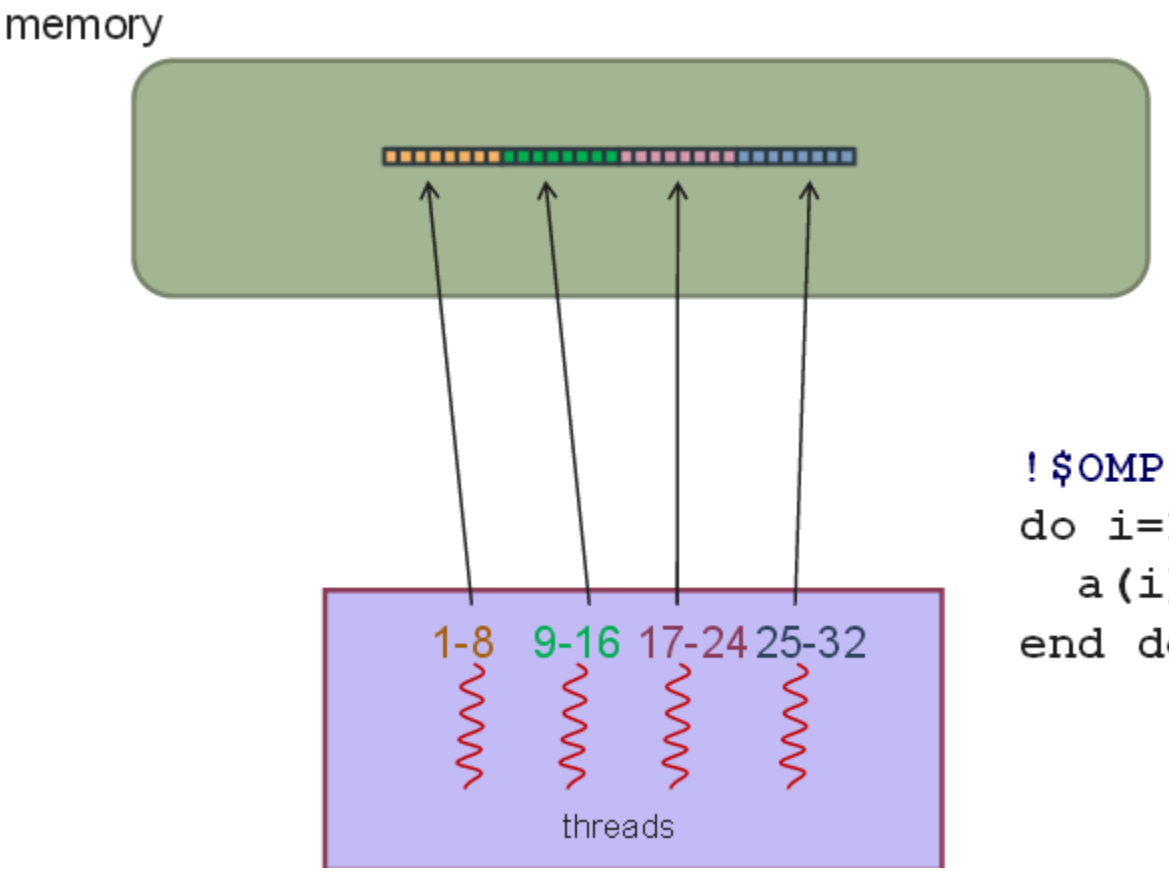
    …often followed by a structured code block
  - Fortran

    `!$omp target` *`directive [clause [,] clause] …]`*

    ...often paired with a matching end directive surrounding a structured code block:

    `!$omp end target` *`directive`*

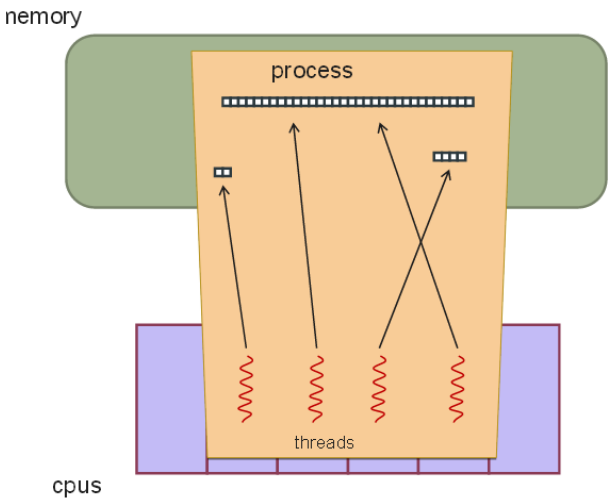
OpenMP: work distribution



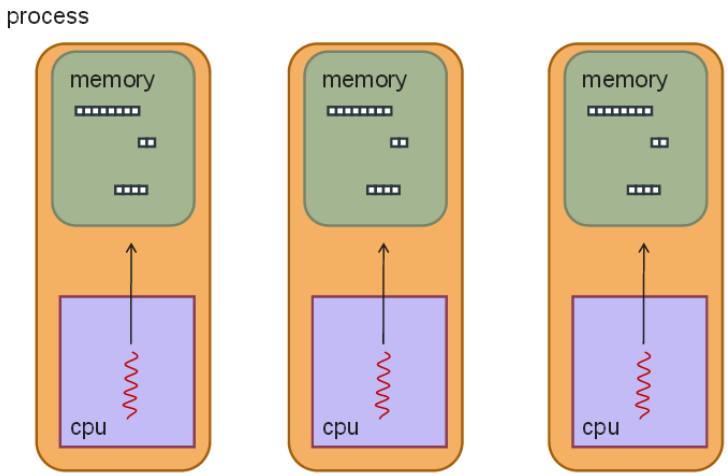

```
!$OMP PARALLEL DO
do i=1,32
  a(i)=a(i)*2
end do
```

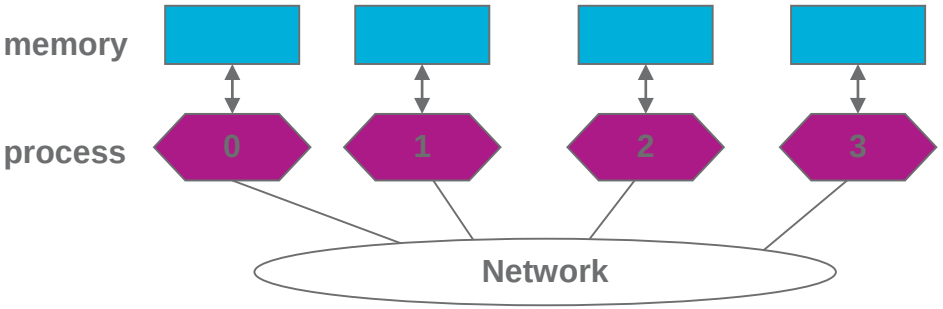
OpenMP memory domain

- 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



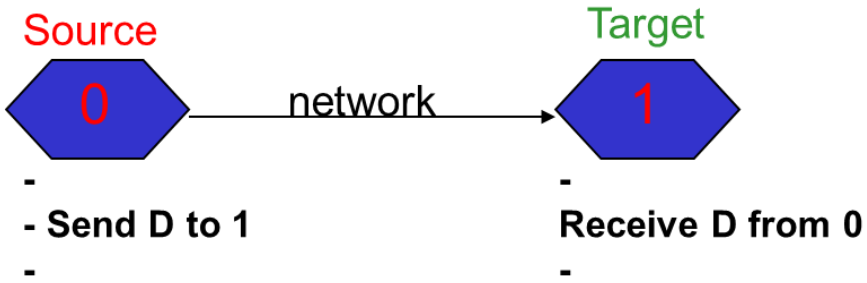
Distributed Memory Message Passing (MPI) model





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

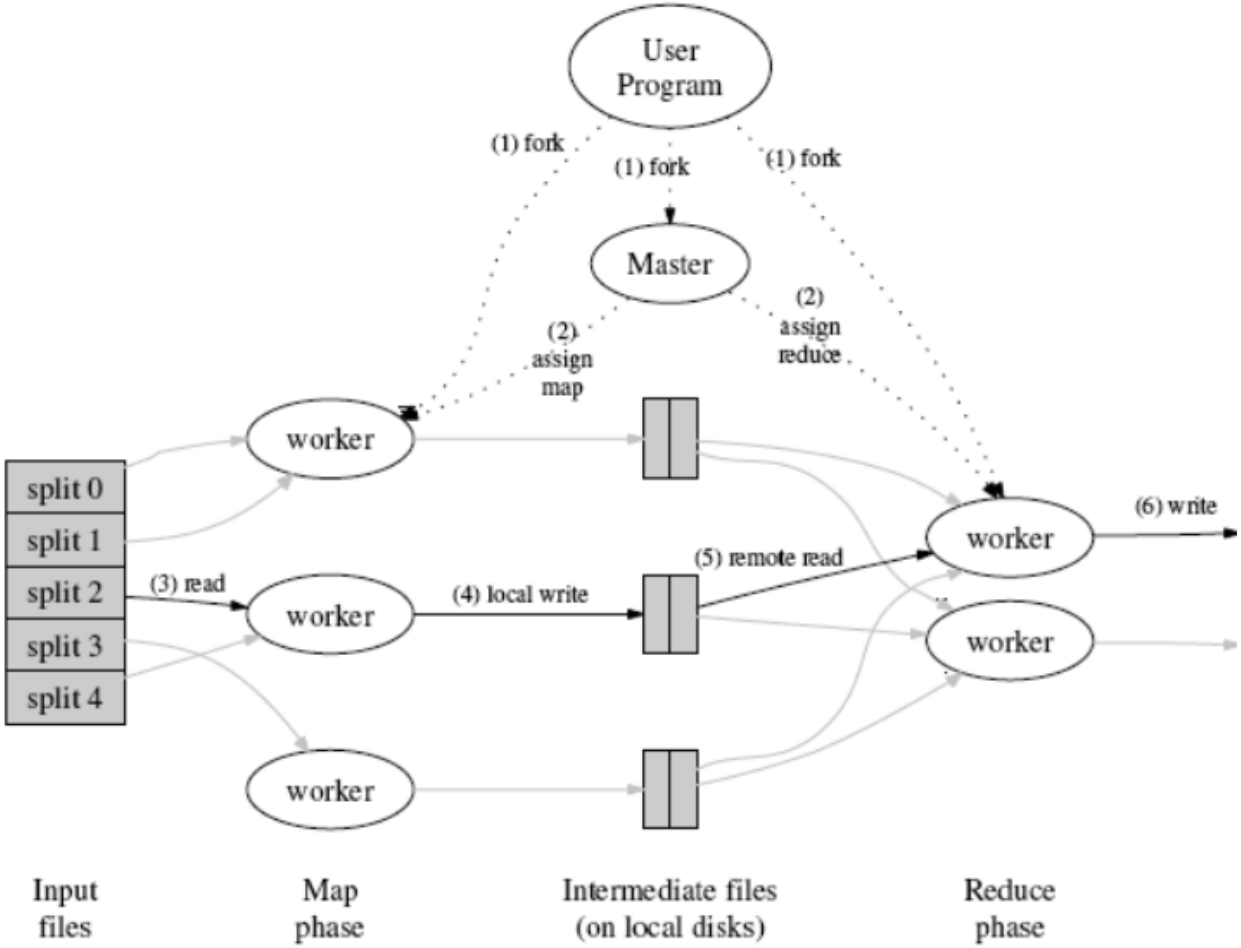
```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
  EmitIntermediate(w, "1");

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.

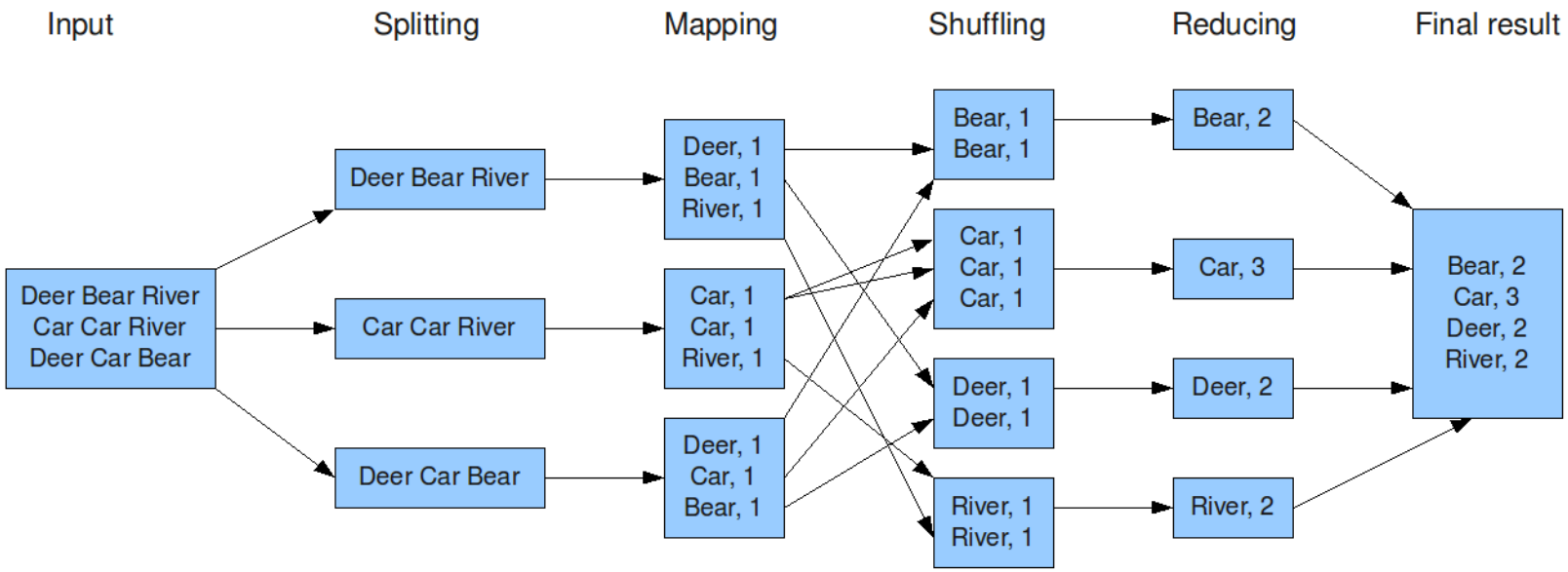
reduce invocations are distributed by partitioning the intermediate key space into R pieces

partitions are specified by user

MapReduce processing scheme



The overall MapReduce word count process



- 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 parallization
    - 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 – Single prec A*X Plus Y in OpenMP - CPU

SAXPY in 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

```
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

```
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

```
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
call saxpy(N, 2.0, x, y)
...
```

SAXPY – Single prec A*X Plus Y in OpenMP – Accelerator (GPU)

SAXPY in C

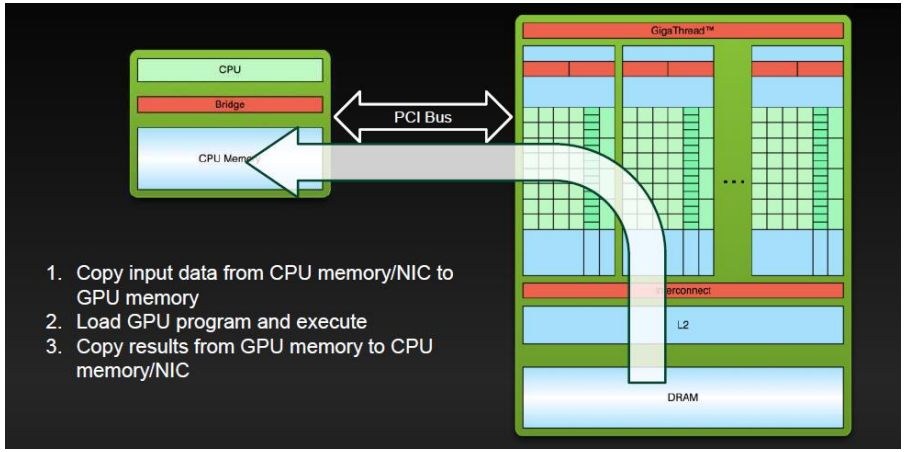
```
void saxpy(int n, float a,
           float *x, float *y)
{
# #pragma omp target teams \
        distribute parallel for
for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



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

SAXPY in Fortran

```
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)
...
```

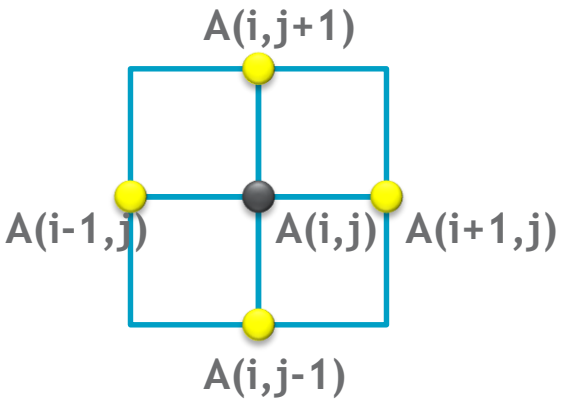
Single example about how to express parallelism and data locality using compiler directives languages using a GPU accelerator

Data must be transfered between CPU and GPU memories



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: $\boldsymbol{\nabla^2 f(x,y) = 0}$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Jacobi Iteration: C Code

```
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++;
}
```

- Iterate until converged
- Iterate across matrix elements
- Calculate new value from neighbors
- Compute max error for convergence
- Swap input/output arrays

Jacobi Iteration: C Code

```
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++;
}
```

Data dependency between iterations.

Independent loop iterations

Independent loop iterations

Identify Parallelism → Express Parallelism → Express Data Locality → Optimize

Jacobi Iteration: OpenMP C Code for CPU

```
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++;
}
```

Parallelize loop across CPU threads

Parallelize loop across CPU threads

Identify Parallelism → Express Parallelism → Express Data Locality → Optimize

Jacobi Iteration: OpenACC C Code – CPU&GPU

```
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++;
}
```

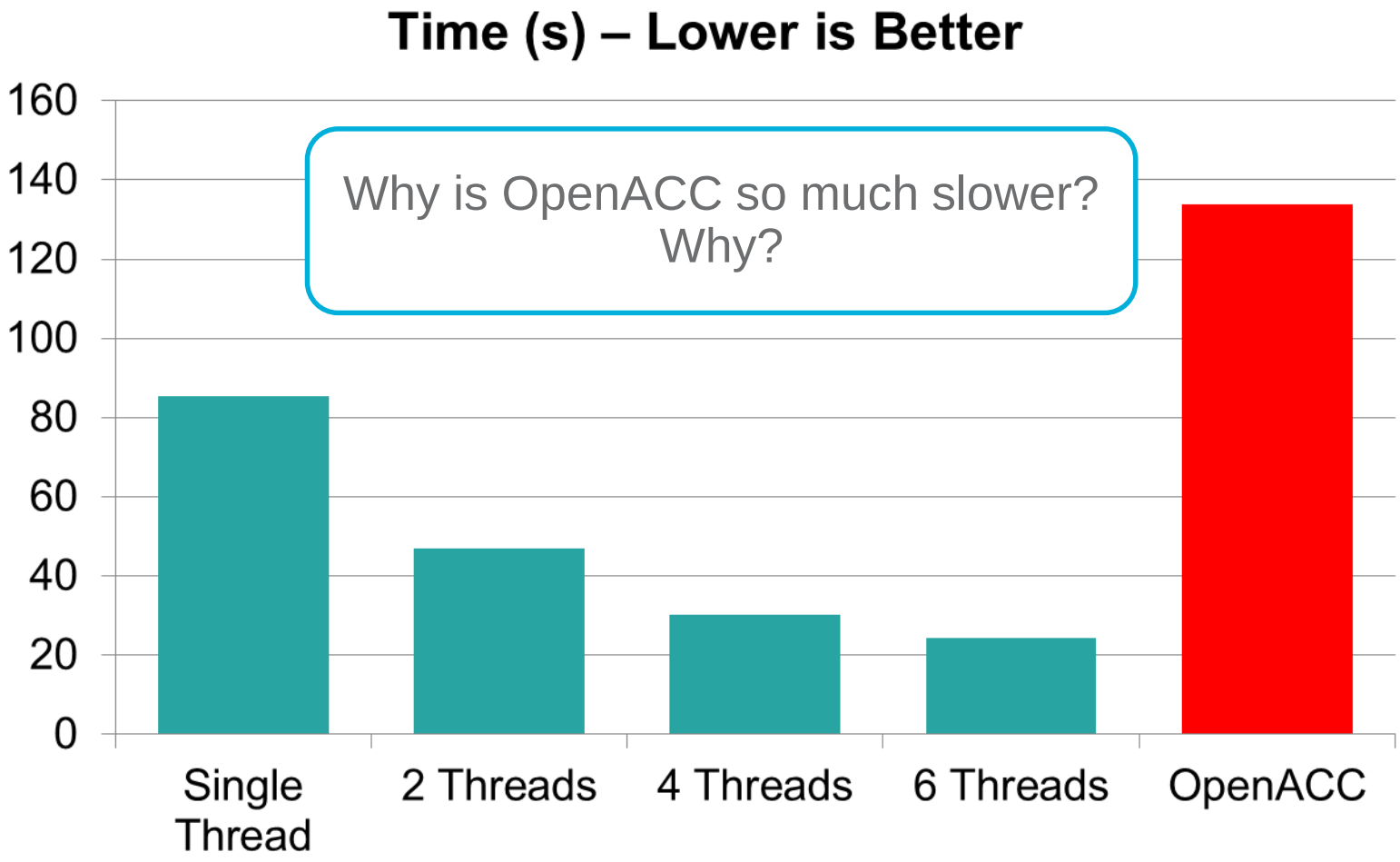
Parallelize loop on accelerator

Parallelize loop on accelerator

Identify Parallelism → Express Parallelism → Express Data Locality → Optimize

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])
         Generating present_or_copyin(A[0:][0:])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     59, Loop is parallelizable
     63, Max reduction generated for error
     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])
         Generating present_or_copyout(A[1:4094][1:4094])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     71, Loop is parallelizable
```



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 memcpy DtoH]
  2.95%  2.61622s      1000  2.6162ms  2.6044ms  2.6319ms  main_56_gpu
  2.39%  2.11884s      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
```

Excessive Data Transfers

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

A, Anew resident on host

Copy

```
#pragma acc parallel loop reduction(max:err)
```

A, Anew resident on accelerator

```
  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]);
    }
  }
```

These copies happen every iteration of the outer while loop!*

Copy

A, Anew resident on accelerator

A, Anew resident on host

```
    ...
}
```

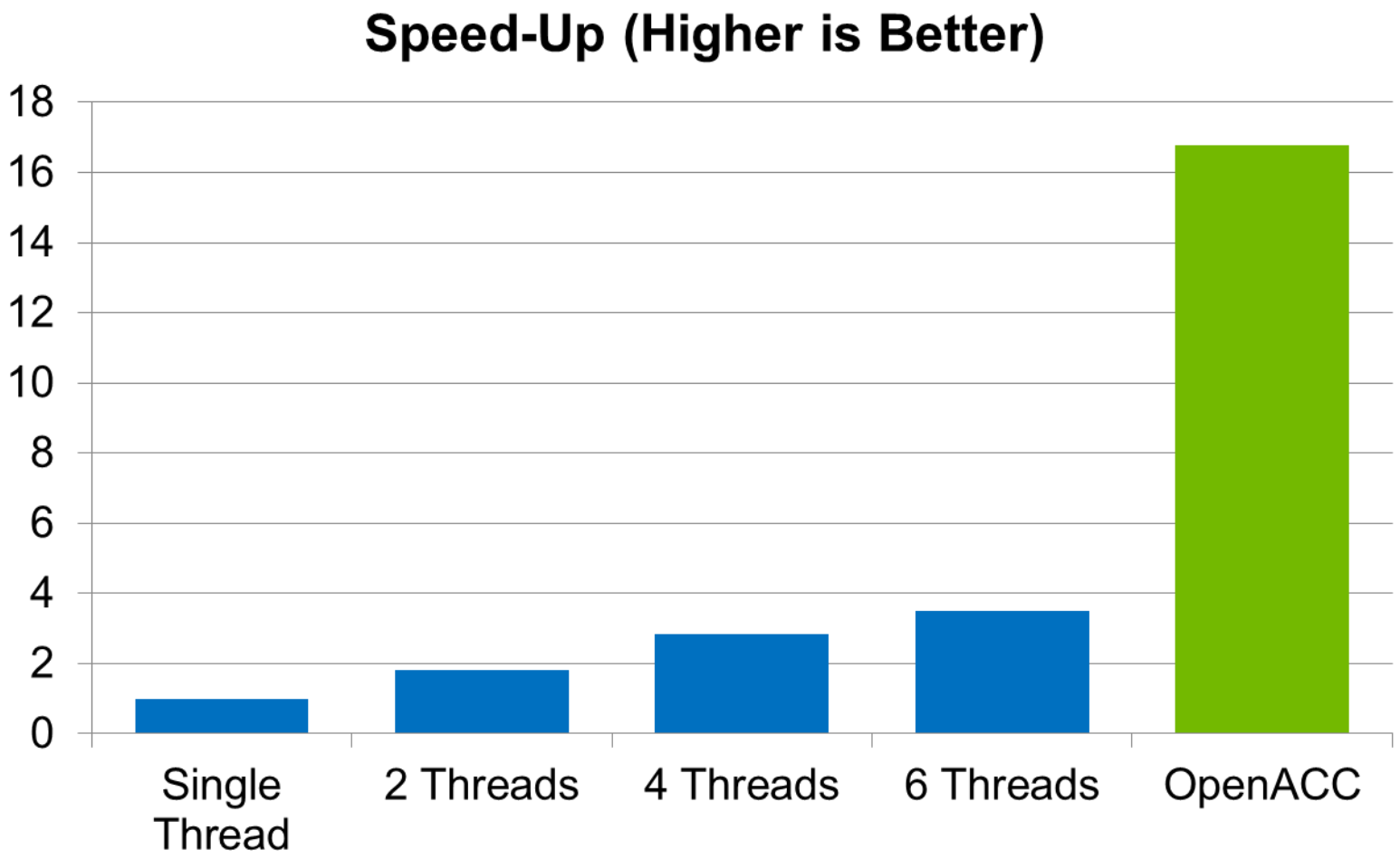
=> Need to use directive to control data location and transfers

Jacobi Iteration: OpenACC C Code

```
#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.

Identify Parallelism → Express Parallelism → Express Data Locality → Optimize



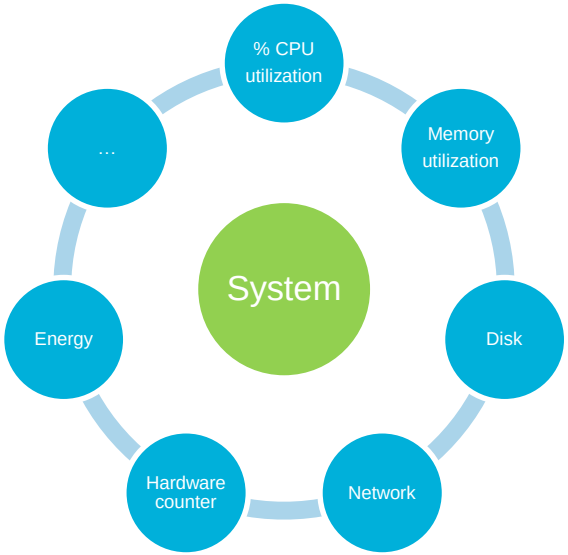
- Programming Languages and Models vs Compilers
  - Accelerator or CPU performance ?

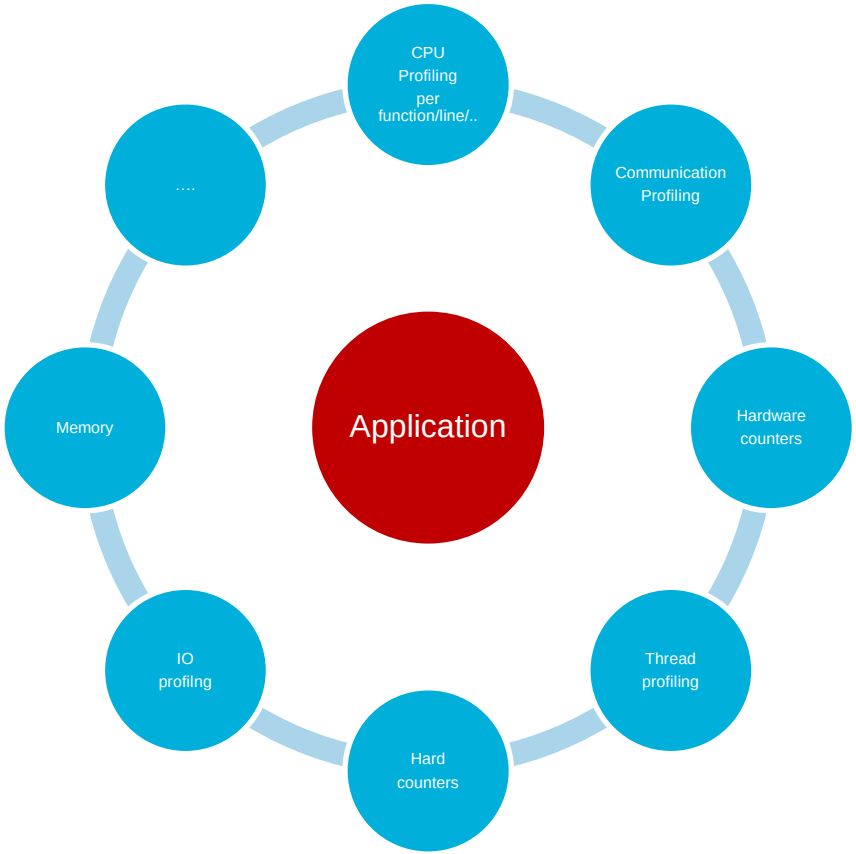
- OpenACC:
  - PGI, (GCC 6.x 7.x), omnicompiler, …

- OpenMP 4.x
  - CLANG, IBM XL compiler, Intel Compiler, …

Performance Analysis Tools

- 2 types of performance monitoring





System 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
  - Locatlity/affinity/
  - …
  - …
- Report + Automated-intelligent assist

top / htop

```
top - 11:11:29 up 21:17,  2 users,  load average: 0.66, 0.62, 0.40
Tasks: 202 total,   3 running, 199 sleeping,   0 stopped,   0 zombie
Cpu0  : 99.0%us,  1.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  : 95.0%us,  5.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  : 95.3%us,  4.7%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  : 94.7%us,  5.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  : 99.3%us,  0.7%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  : 99.3%us,  0.7%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  24660408k total,   711064k used, 23949344k free,    98648k buffers
Swap:  1052248k total,        0k used,  1052248k free,   326752k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7516 [redacted] 25   0  162m  35m 2196 R 799.7  0.1   1:14.30 poisson.exe.x86
    1 root      15   0 10344  680  568 S  0.0  0.0   0:01.12 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:00.01 migration/0
```

- System monitoring
  - Core usage
  - Memory usage
  - Process information
    - Running status
    - Owner
- Monitor the node
  - Limited by operating system

IBM Client Center

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

```
nmon-16a──────[H for help]──Hostname=vm26──────────Refresh= 2secs ──08:53.14─

  ──────────────────────────────
   _ __  _ __ ___   ___  _ __
  | '_ \| '_ ` _ \ / _ \| '_ \
  | | | | | | | | | (_) | | | |
  |_| |_|_| |_| |_|\___/|_| |_|

  ──────────────────────────────

                                  For help type H or ...
                                   nmon -?  - hint
                                   nmon -h  - full details

                                  To stop nmon type q to Quit

  DISTRIB_DESCRIPTION="Ubuntu 15.04"
  PowerVM POWER8 (architected), altivec supported CHRP IBM,8284-22A
  PowerVM Entitlement=1.00   VirtualCPUs=2 LogicalCPUs=16
  PowerVM SMT=8 Capped=0
  Processor Clock=3425.000000MHz              Little Endian

  Use these keys to toggle statistics on/off:
    c = CPU         l = CPU Long-term     - = Faster screen updates
    m = Memory      V = Virtual memory    + = Slower screen updates
    d = Disks       n = Network           j = File Systems
    r = Resource    N = NFS               . = only busy disks/procs
    k = Kernel      t = Top-processes     h = more options
                                          q = Quit
```

```
File  Edit  View  Search  Terminal  Help
nmon-16b──────[H for help]──Hostname=nag-desktop──Refresh= 2secs ──23:19.50─
 CPU Utilisation
---------------------------+-------------------------------------------------+
CPU User%  Sys% Wait%  Idle|0          |25         |50         |75       100|
  1  42.7   6.5  50.8   0.0|UUUUUUUUUUUUUUUUUUUUSSSWWWWWWWWWWWWWWWWWWWWWWWWWW>
  2  20.5   2.5   0.0  77.0|UUUUUUUUUUS                                      >
  3  21.8   1.0   0.0  77.2|UUUUUUUUUU                                      > |
  4   9.0   1.5  89.5   0.0|UUUUWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW>|
---------------------------+-------------------------------------------------+
Avg  23.5   2.9  35.2  38.4|UUUUUUUUUUUSWWWWWWWWWWWWWWWWWW             >      |
---------------------------+-------------------------------------------------+
 CPU Utilisation Wide View
100%-+CPU(s)=4+---------+---------+---------+---------+---------+-----+100%
 90%-|#   |                                                            |-90%
 80%-|#   |                                                            |-80%
 70%-|#   |                                                            |-70%
 60%-|#   |                                                            |-60%
 50%-|#. .|                                                            |-50%
 40%-|### |                                                            |-40%
 30%-|### |                                                            |-30%
 20%-|####|                                                            |-20%
 10%-|####|                                                            |-10%
 CPU +1--------+10-------+20-------+30-------+40-------+50-------+60--+--0%
 CPU Utilisation Stats
ALL     93.1    0.0   11.0  152.4  139.5    0.0    0.5    0.0    0.0    0.0
CPU   User%  Nice%   Sys%  Idle%  Wait% HWirq% SWirq% Steal% Guest% GuestNice%
  1    42.3    0.0    6.0    0.0   50.3    0.0    0.5    0.0    0.0    0.0
  2    20.4    0.0    2.5   76.7    0.0    0.0    0.0    0.0    0.0    0.0
  3    21.4    0.0    1.0   75.7    0.0    0.0    0.0    0.0    0.0    0.0
  4     9.0    0.0    1.5    0.0   89.2    0.0    0.0    0.0    0.0    0.0
 Disk I/O ──/proc/diskstats──mostly in KB/s──────Warning:contains duplicates─
DiskName Busy  Read WriteKB|0          |25         |50         |75       100|
mmcblk0  100%   21.9 3606.4|                                                >
mcblk0p1   0%    0.0    0.0|>                                               |
mcblk0p2 100%   21.9 3606.4|                                                >
Totals Read-MB/s=0.0      Writes-MB/s=7.0      Transfers/sec=22.9
```

Application performance analysis tools

- Sampling vs instrumented instrumentation
  - Sampling limited overheard
  - 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

- Step1 : 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.
%    cumulative self          self   total
time seconds    seconds calls s/call s/call name
33.86 15.52     15.52    1    15.52  15.52  func2
33.82 31.02     15.50    1    15.50  15.50  new_func1
33.29 46.27     15.26    1    15.26  30.75  func1
0.07  46.30     0.03                        main

% the percentage of the total running time of the
time program used by this function.
```

perf Linux command

- 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_otpions] mpyperf.sf execution [args]
- cat myperf.sh

```
#!/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 bacth 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)

Command	Purpose
valgrind <program>	Perform regular memory checking
Valgrind –v <program>	Verbose mode
valgrind --leak-check=full <program>	Perform memory leak checking

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:

```
Communication Activity by actual args
Collectives
Operation        Context  Algo  Comm size    Message size      Calls  Cost(%)
-----------------------------------------------------------------------------
Allreduce
1                     58     1          4              24          1     0.00
2                     58     1          4               4          8     0.00
3                     58     1          4               8         12     0.03
4                     58     1          4            1376        181     0.04
5                     58     1          4            1344         19     0.01
6                     58     1          4            1216          1     0.00
7                     58     1          4             224          1     0.00
8                      0     5        192               8          2     0.00
9                      0     5        192             968          1     0.00
10                     0     5        192             288          2     0.01
11                     0     5        192             768          2     0.00
Barrier
1                     62     5        160               0          1     0.00
2                      0     5        192               0          1     0.00
Bcast
...
Gather
1                     52     3          5              32         25     0.01
2                     54     3          4              36         25     0.00
3                     56     3          8              28         25     0.01
Reduce
1                     60     1         40              24          1     0.00
2                     60     1         40               4          8     0.00
3                     60     1         40               8         12     0.01
4                     60     1         40            1376        181     0.21
5                     60     1         40            1344         19     0.03
6                     60     1         40            1216          1     0.00
7                     60     1         40             224          1     0.00
Scatter
1                     62     1        160               8          1     0.00
Scatterv
1                     62     1        160          315840          2     0.03
2                     62     1        160           52640          1     0.08
=============================================================================
```

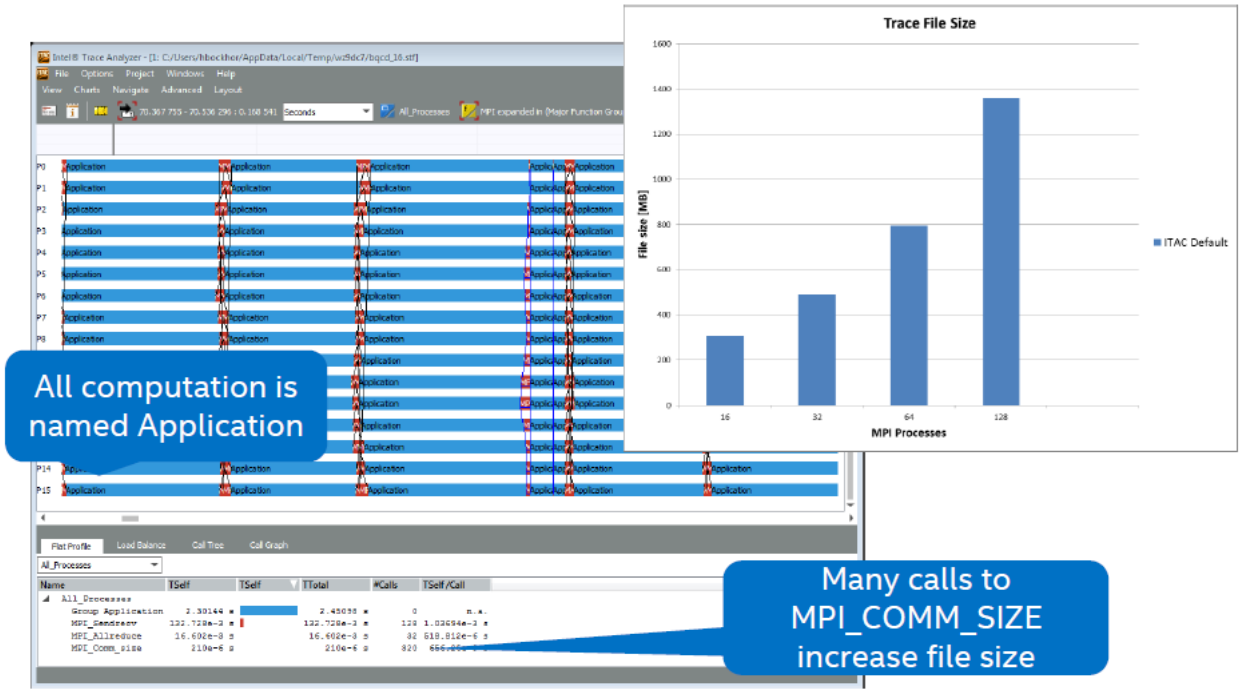
```
$ export I_MPI_STATS=3; export I_MPI_STATS_SCOPE=coll
```

INTEL MPI Profiling: ITAC (~vampire, TAU, …)

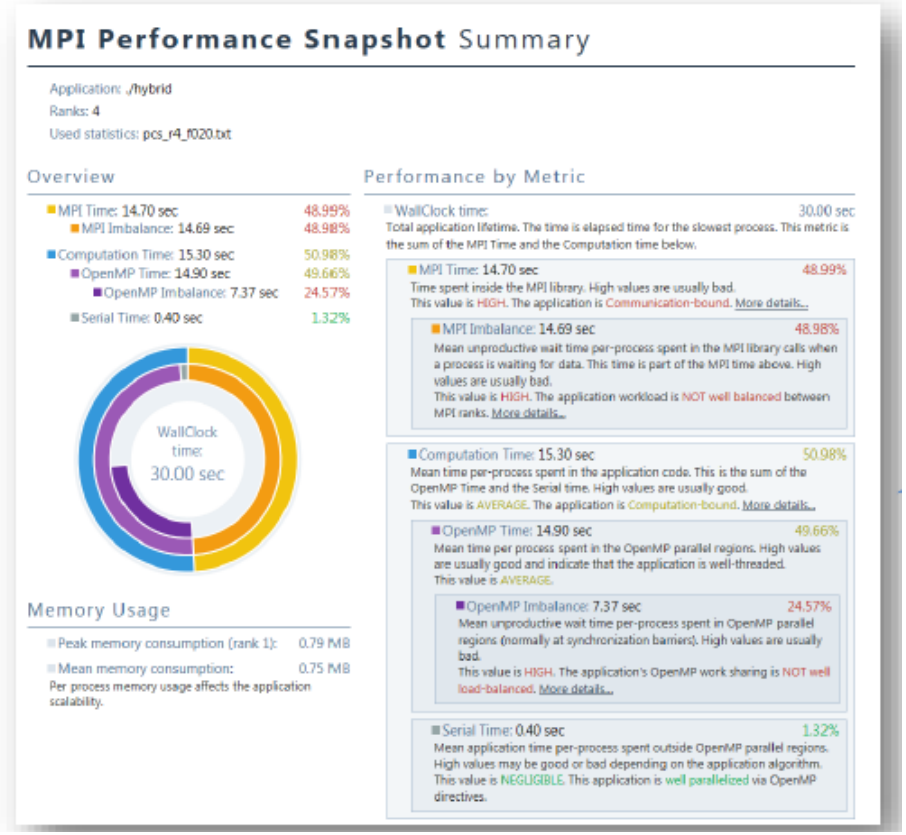
Start with simple default MPI only trace: `mpirun -trace …`

Full instrumentation using `-tcollect`

Simple MPI Trace – Trace File Size



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

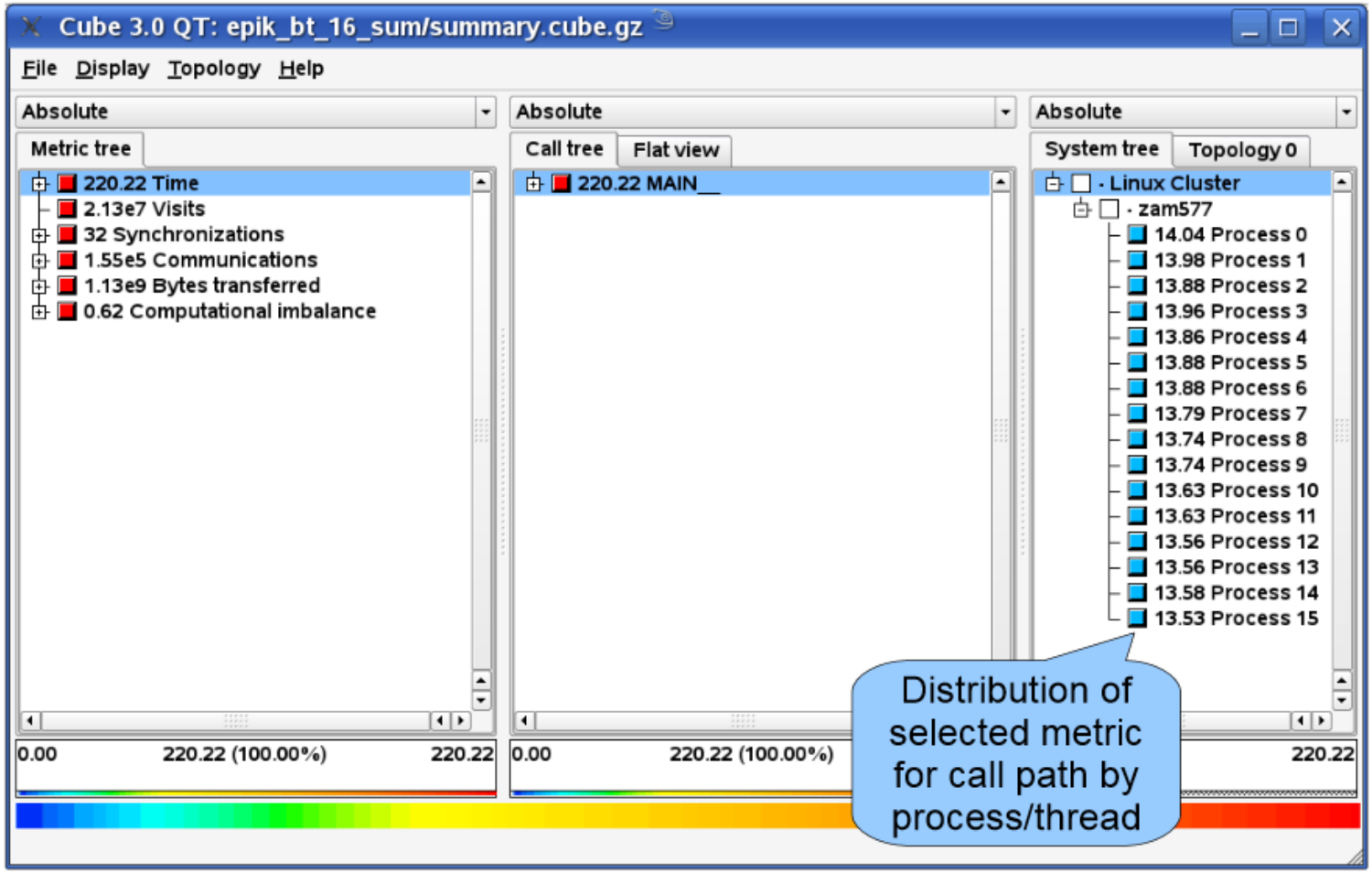
Scalasca (http://www.scalasca.org/) – open source

- 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.

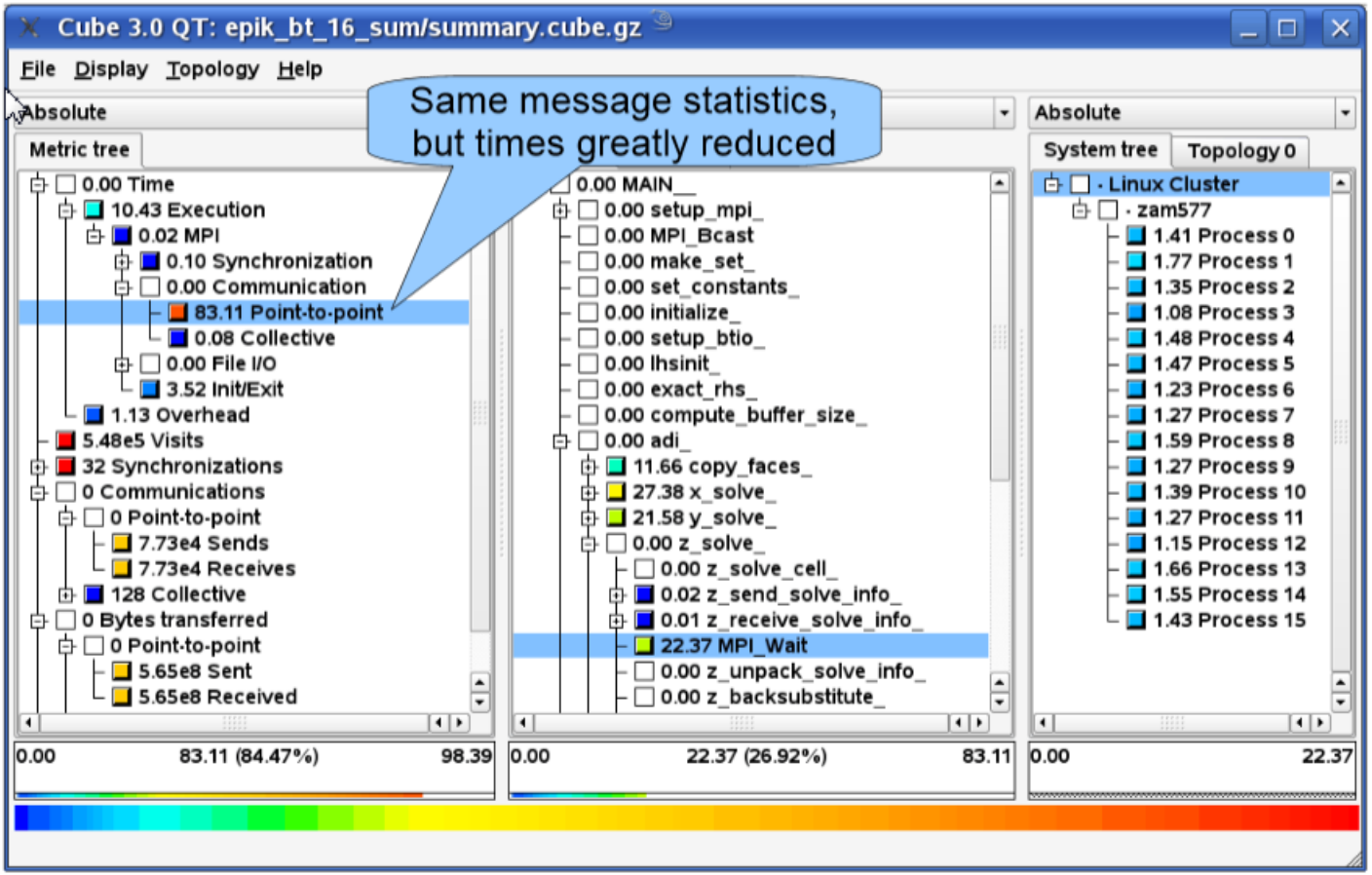
- Perf

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

Scalasca analysis report exploration 1/2



Scalasca analysis report exploration 2/2



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