Lecture 8

S4 - Core Distributed Middleware Programming in JEE

presentation

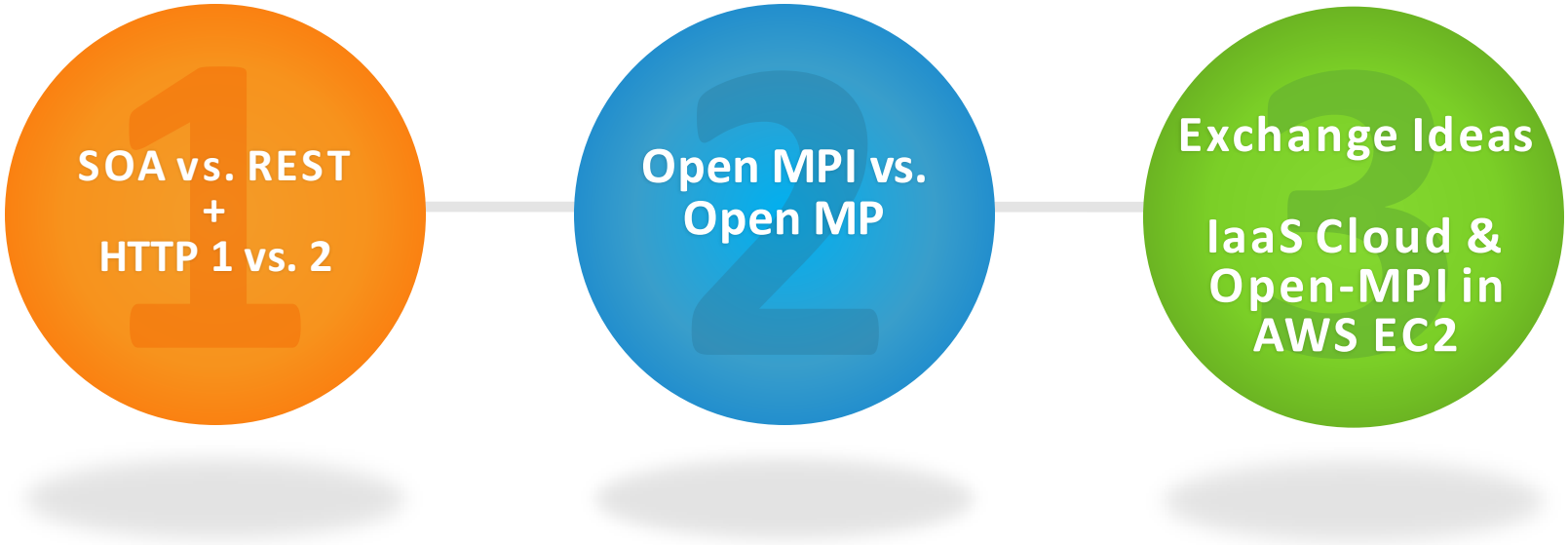
DAD – Distributed Applications Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 8

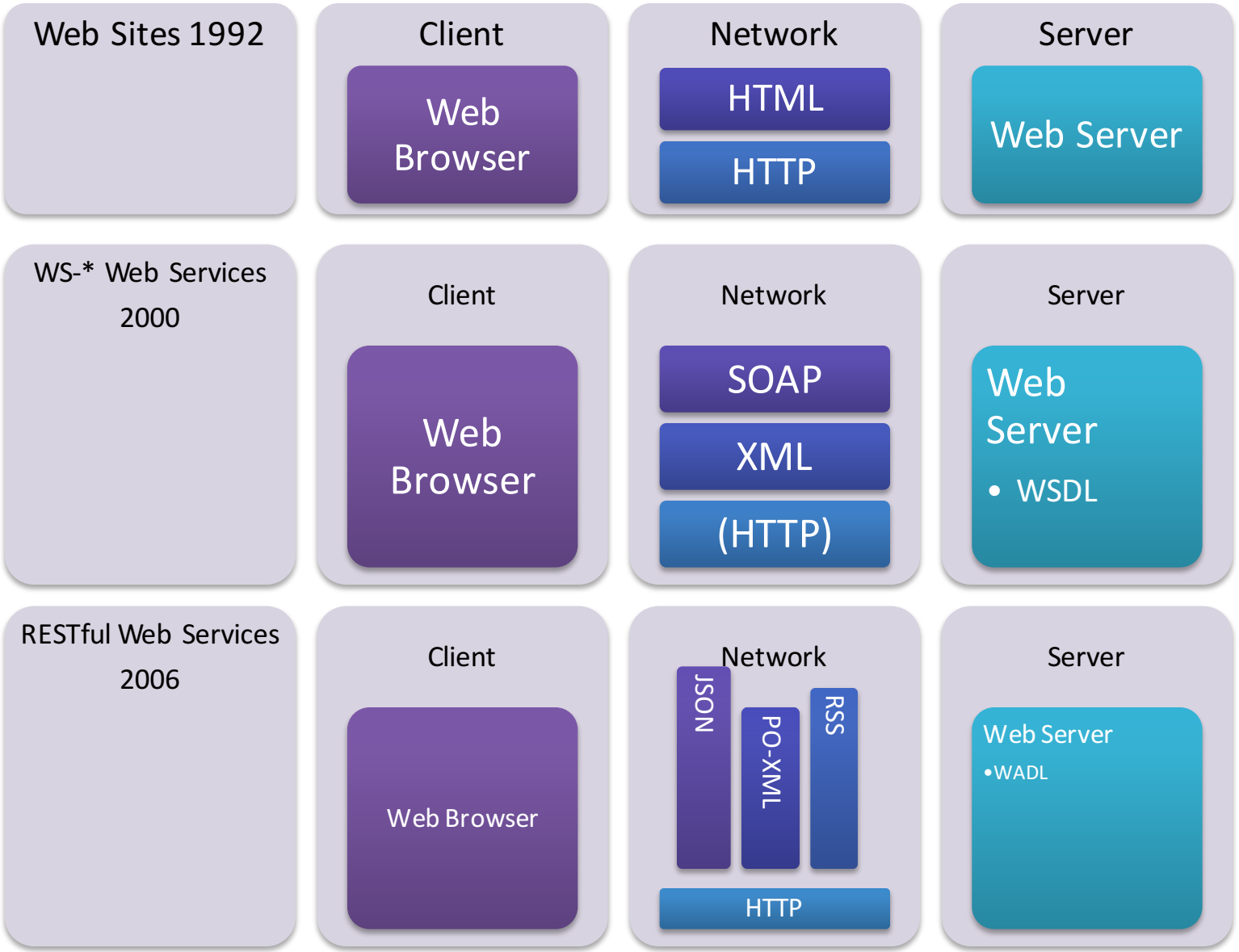


REST and HTTP v2

REST vs. Web Services

HTTP 2 vs. HTTP 1

1. Communications Protocols HTTP-REST, CoAP, MQTT

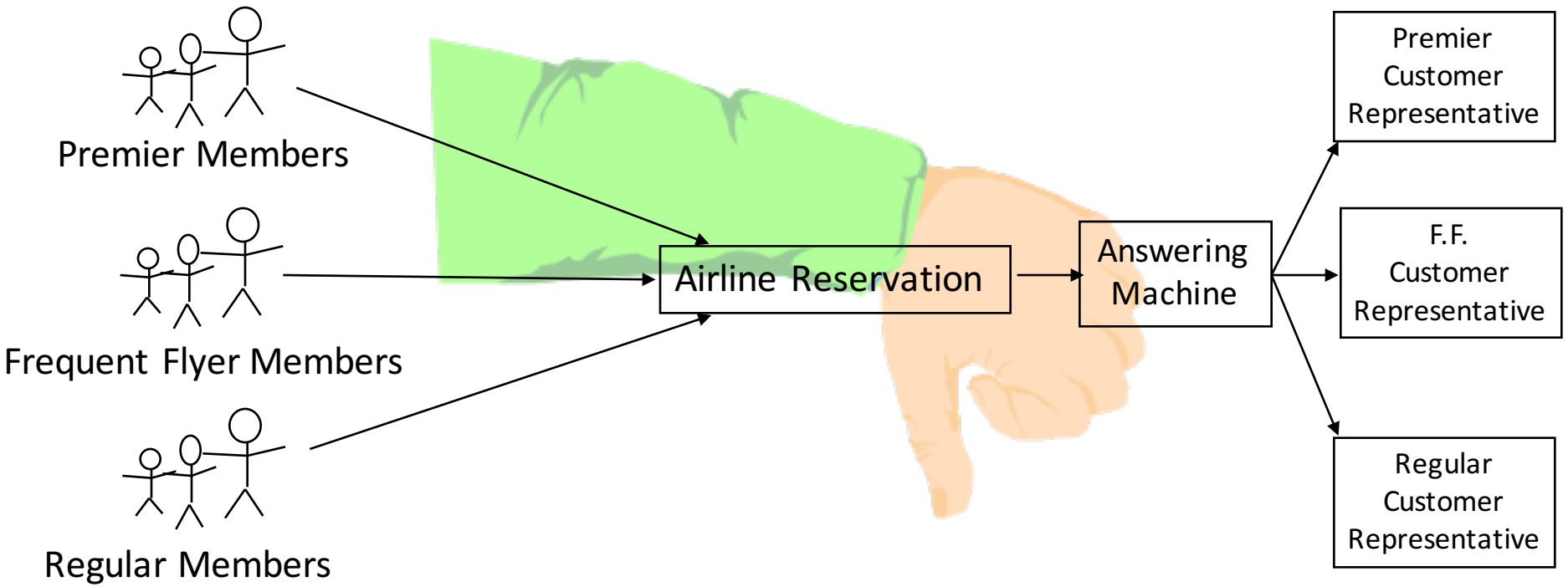


1. Communications Protocols HTTP-REST, CoAP, MQTT

WS-* vs. REST Comparison

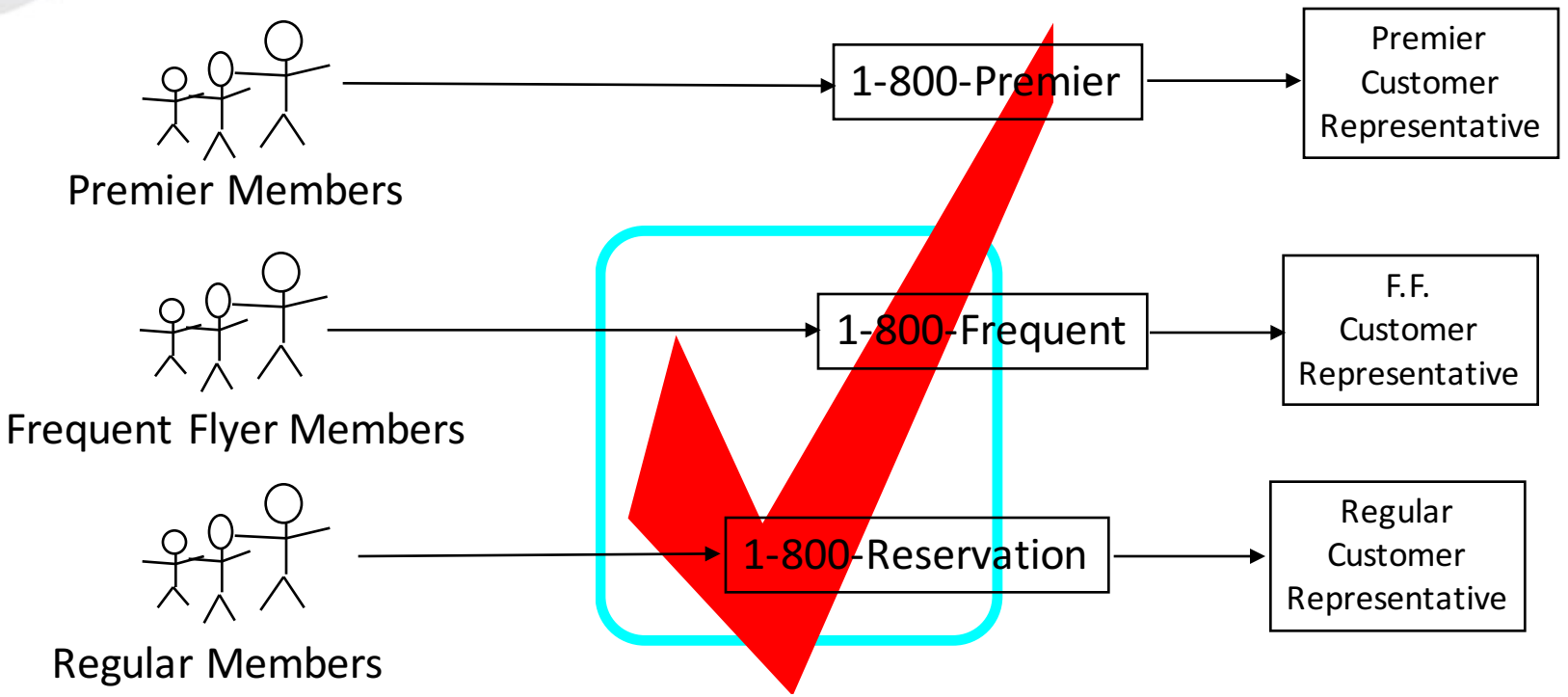
1. Communications Protocols HTTP-REST vs. WS Recap

This Ain't the REST Design Pattern



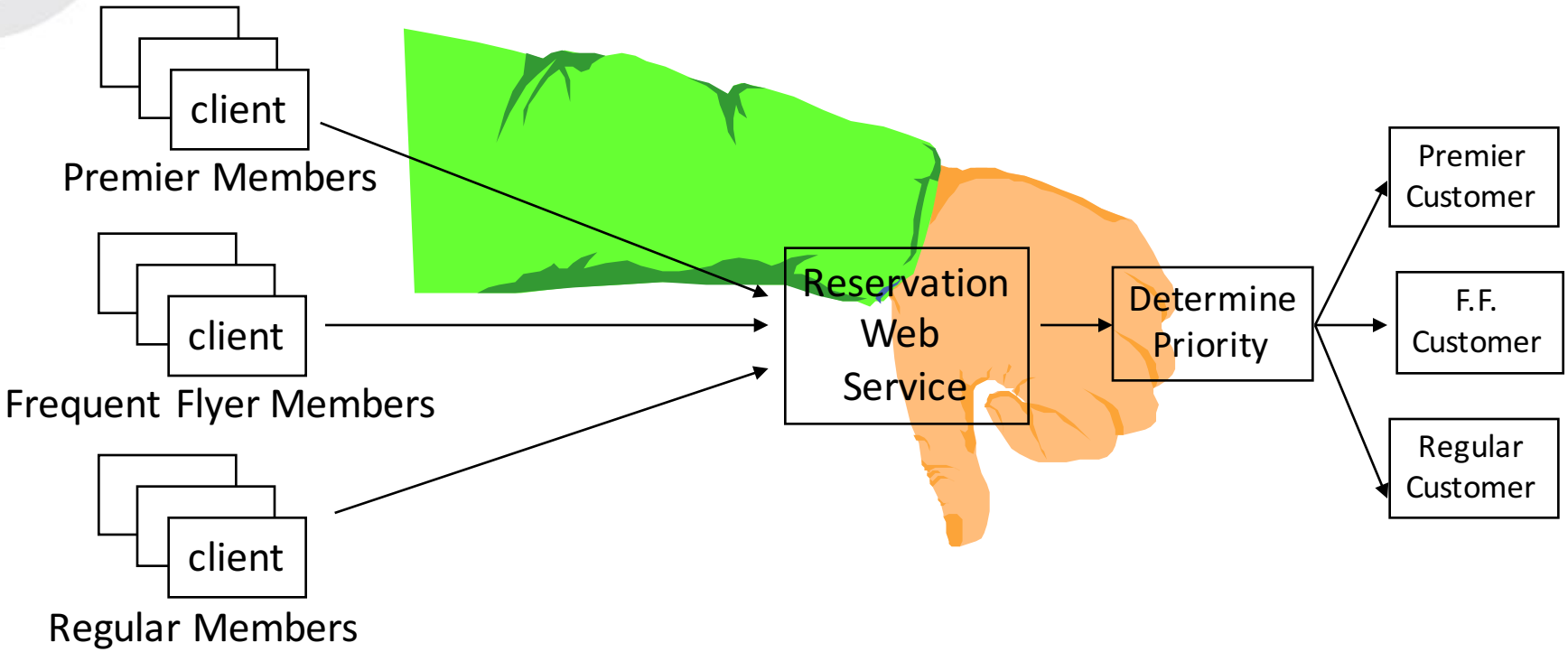
1. Communications Protocols HTTP-REST vs. WS Recap

This is the REST Design Pattern



1. Communications Protocols HTTP-REST vs. WS Recap

This ain't the REST Design Pattern

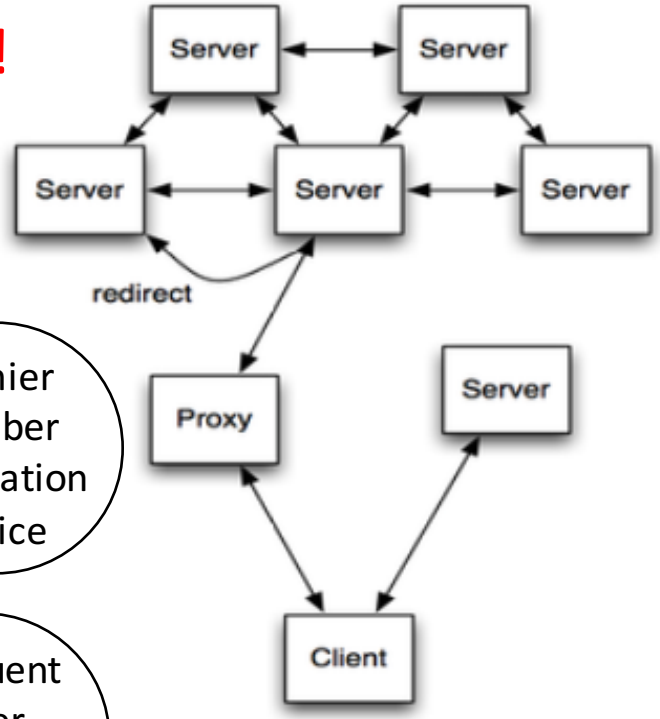


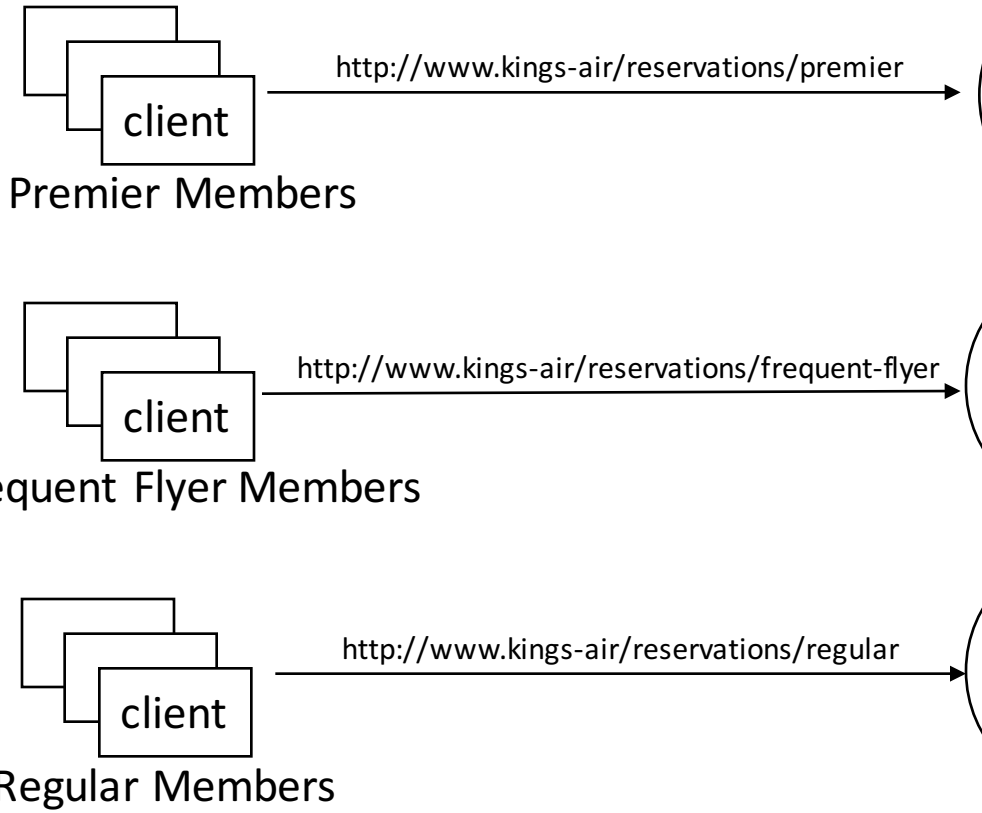
1. Communications Protocols HTTP-REST vs. WS Recap

REST Approach in Web Architecture:

URLs are Cheap! Use Them!

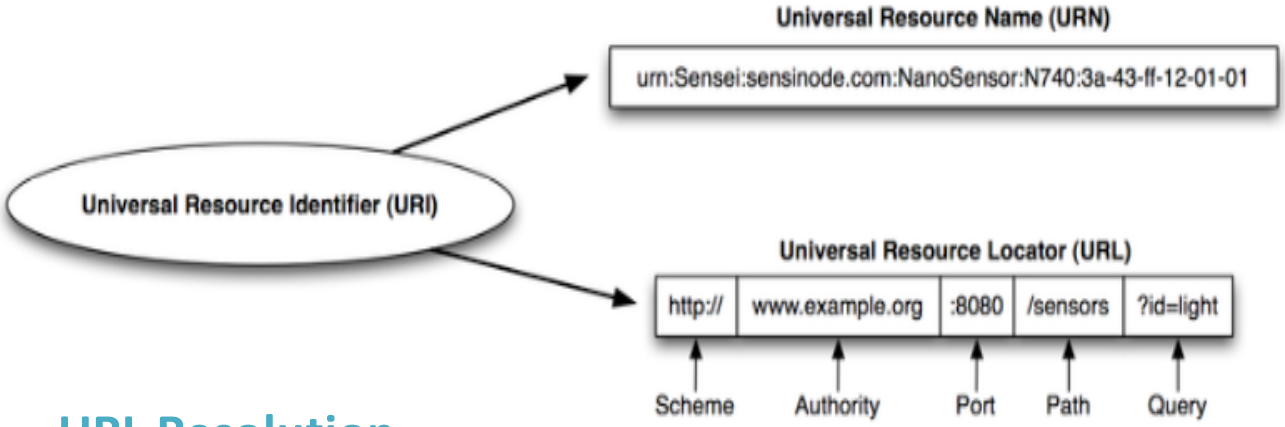
The airline provides several URLs - one URL for premier members, a different URL for frequent flyers, and still another for regular customers.



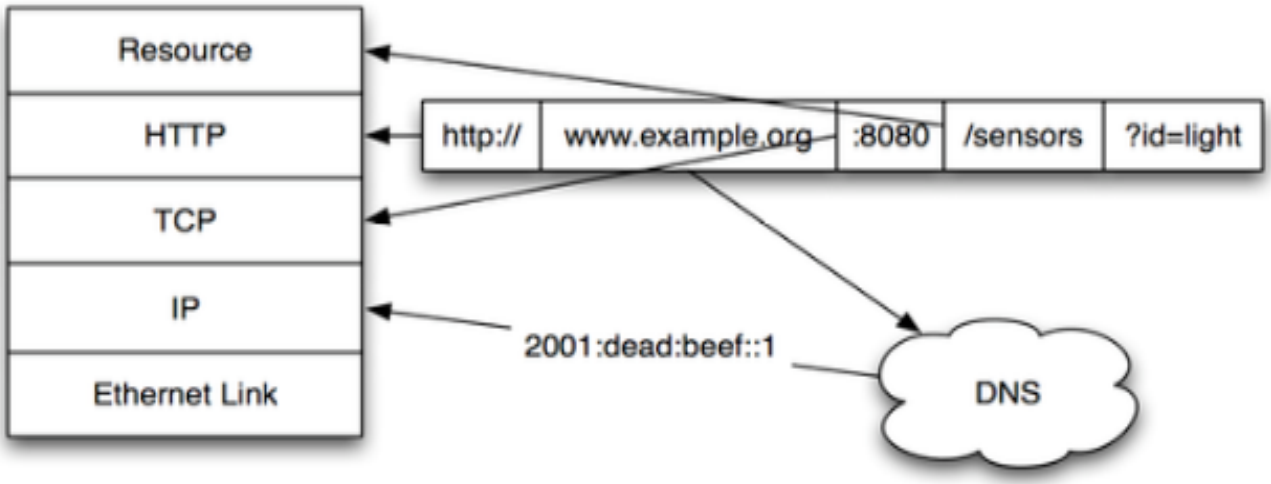


1. Communications Protocols HTTP-REST Recap

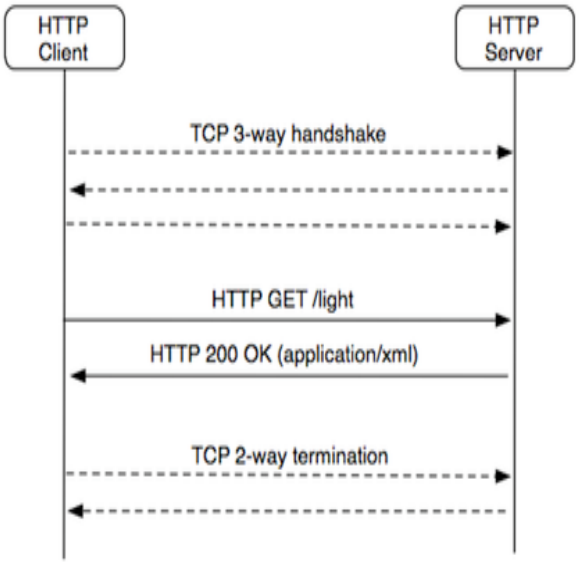
Web Naming



URL Resolution



An HTTP Request



See RFC2616 - Hypertext Transfer Protocol v1.1

1. Communications Protocols HTTP-REST vs. WS Recap

- Simple **web service** as an example: querying a phonebook application for the details of a given user

- Using Web Services and SOAP, the request would look something like this:

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
        <soap:body
  pb="http://www.acme.com/phonebook">
                <pb:GetUserDetails>
                <pb:UserID>12345</pb:UserID>
                </pb:GetUserDetails>
        </soap:Body>
</soap:Envelope>
```

- Simple **REST service** as an example

- And with REST? The query will probably look like this:

  http://www.acme.com/phonebook/UserDetails/12345

- GET /phonebook/UserDetails/12345 HTTP/1.1

  Host: www.acme.com

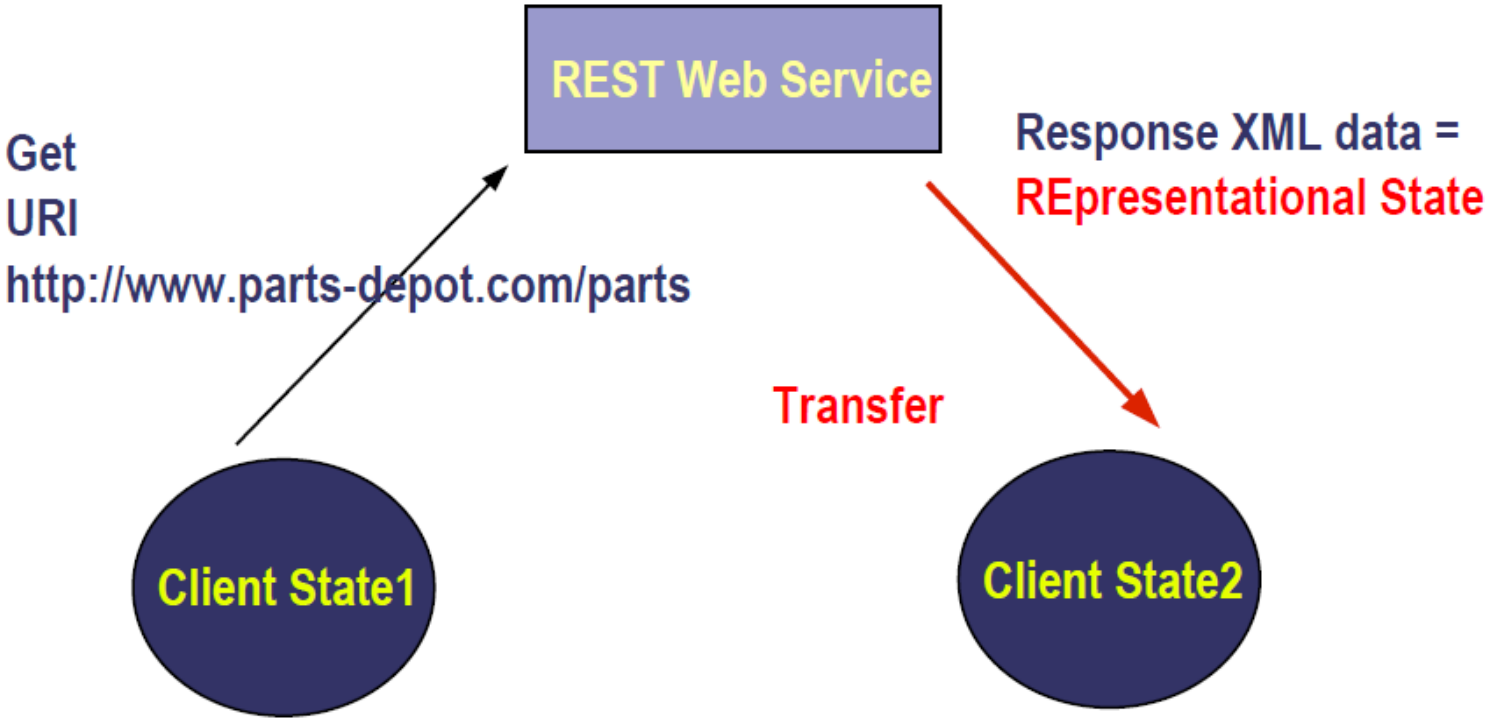
  Accept: application/xml

- Complex query:

  http://www.acme.com/phonebook/UserDetails?firstName=John&lastName=Doe

What is REST?

REpresentational State Transfer



HTTP Request/Response As REST

http://www.javapassion.com/webservices/RESTPrimer.pdf

1. Communications Protocols HTTP-REST Recap

REST over HTTP – Uniform interface

- CRUD operations on resources
  - Create, Read, Update, Delete
- Performed through HTTP methods + URI

Operation	SQL	HTTP
Create	INSERT	PUT / POST
Read (Retrieve)	SELECT	GET
Update (Modify)	UPDATE	PUT / PATCH
Delete (Destroy)	DELETE	DELETE

CRUD Operations	4 main HTTP methods	
	Verb	**Noun**
Create (Single)	POST	Collection URI
Read (Multiple)	GET	Collection URI
Read (Single)	GET	Entry URI
Update (Single)	PUT	Entry URI
Delete (Single)	DELETE	Entry URI

POST = Create
GET = Read
PUT = Update
DELETE = Delete

http://www.javapassion.com/webservices/RESTPrimer.pdf

HTTP/2 vs. HTTP/1

In 2015, however, the Internet Engineering Task Force (IETF) released HTTP/2, the second major version of the most useful internet protocol, HTTP.

SPDY as a precursor to HTTP/2

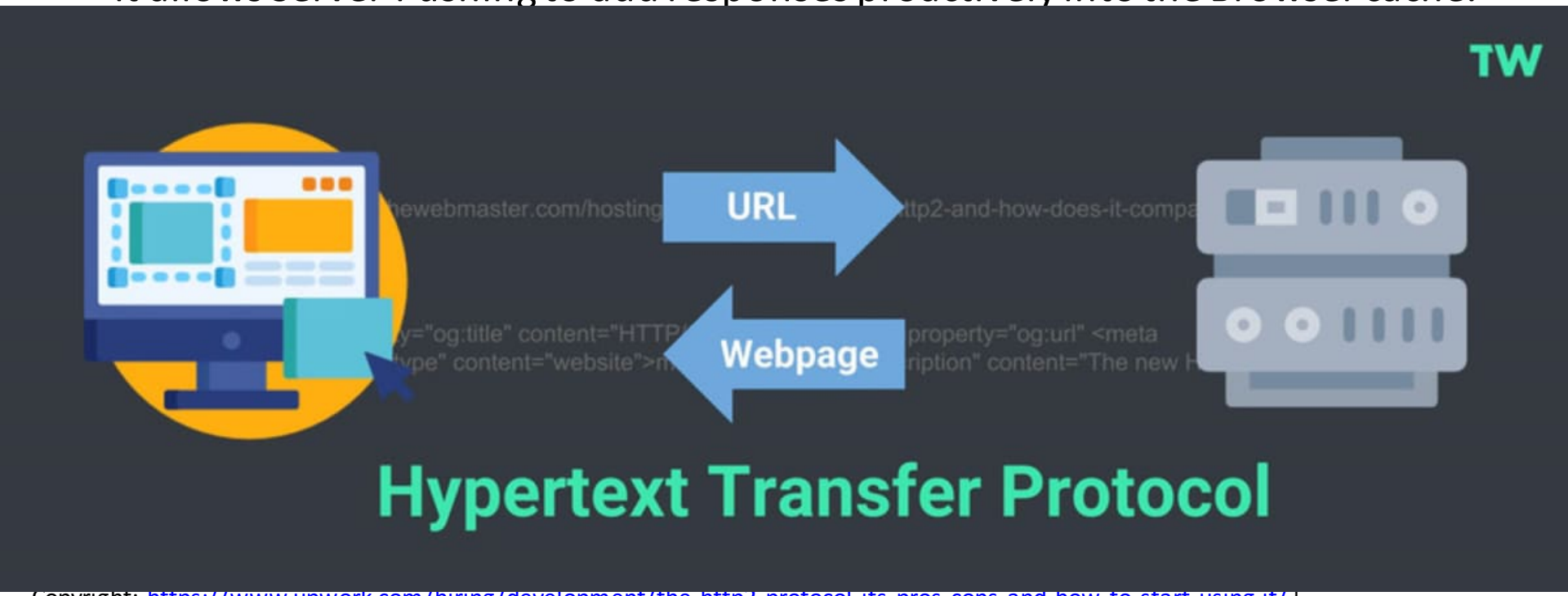
Google was the first to investigate issues with HTTP/1.1. At the time, they were spending millions of dollars a year to support their data centers, and the HTTP/1.1 protocol simply cost too much in terms of CPU resources and internet connection capacity. They developed SPDY as an experimental alternative to HTTP/1.1—a protocol designed for better security and improved page load times that would be the precursor to HTTP/2. **Pros for HTTP/2:**

- **HTTP/2 is binary, instead of textual.**
- **HTTP/2 is fully multiplexed.**
- It uses **header compression** HPACK to reduce overhead.
- **It allows servers to "push" responses** proactively into client cache
- It uses the new **ALPN extension** which allows for faster encrypted connection
- It **reduces additional round trip times (RTT)**, making your website load faster without any optimization.
- **Domain sharding** and asset concatenation are no longer needed with HTTP/2.
- It's **widely supported by browsers**. ***HTTP 3** - The new HTTP/3, based on the QUIC protocol, is anticipated to be released in late 2019.

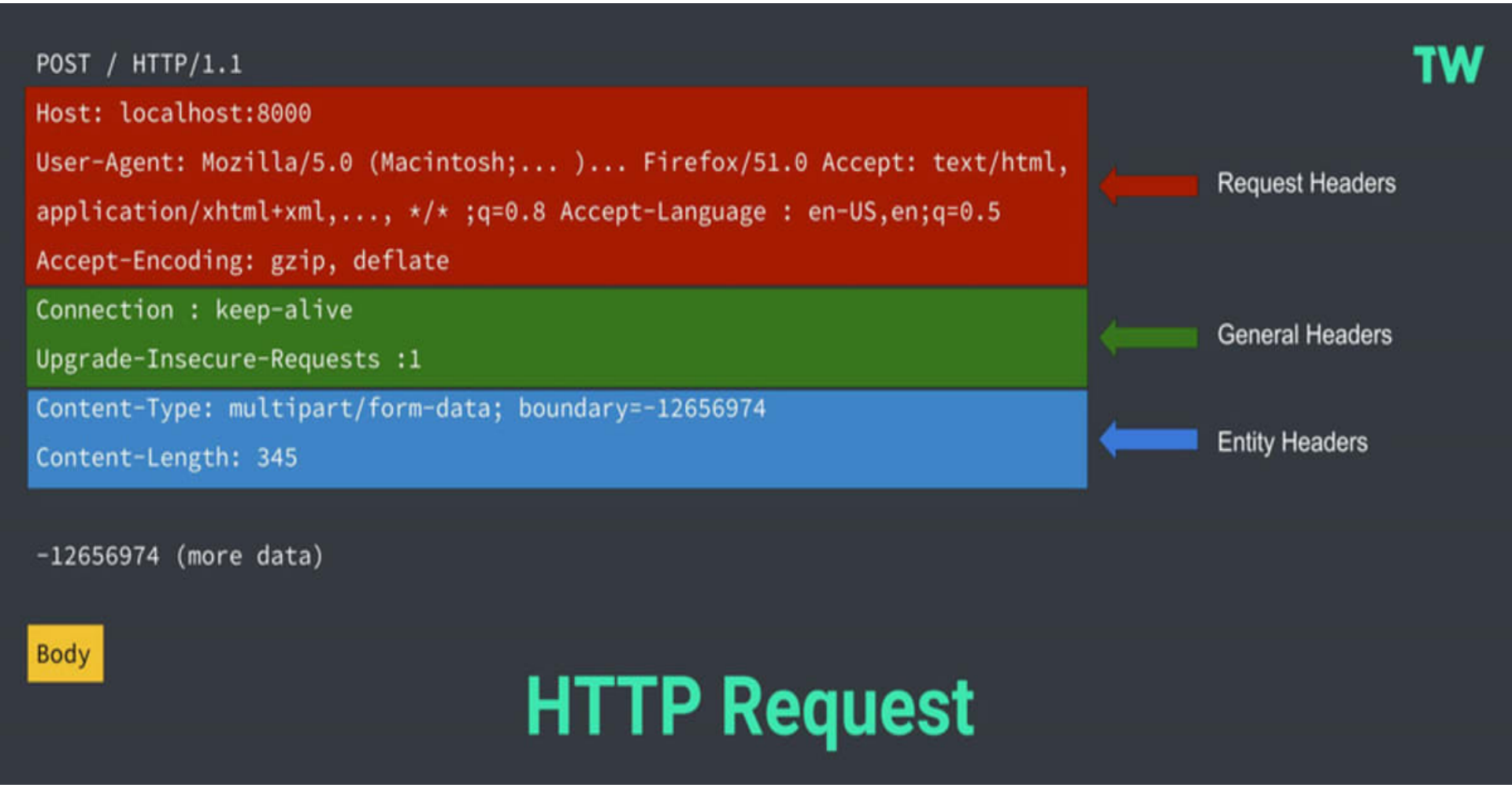
HTTP/2 vs. HTTP/1

The key differences HTTP/2 has to HTTP/1.x are as follows:

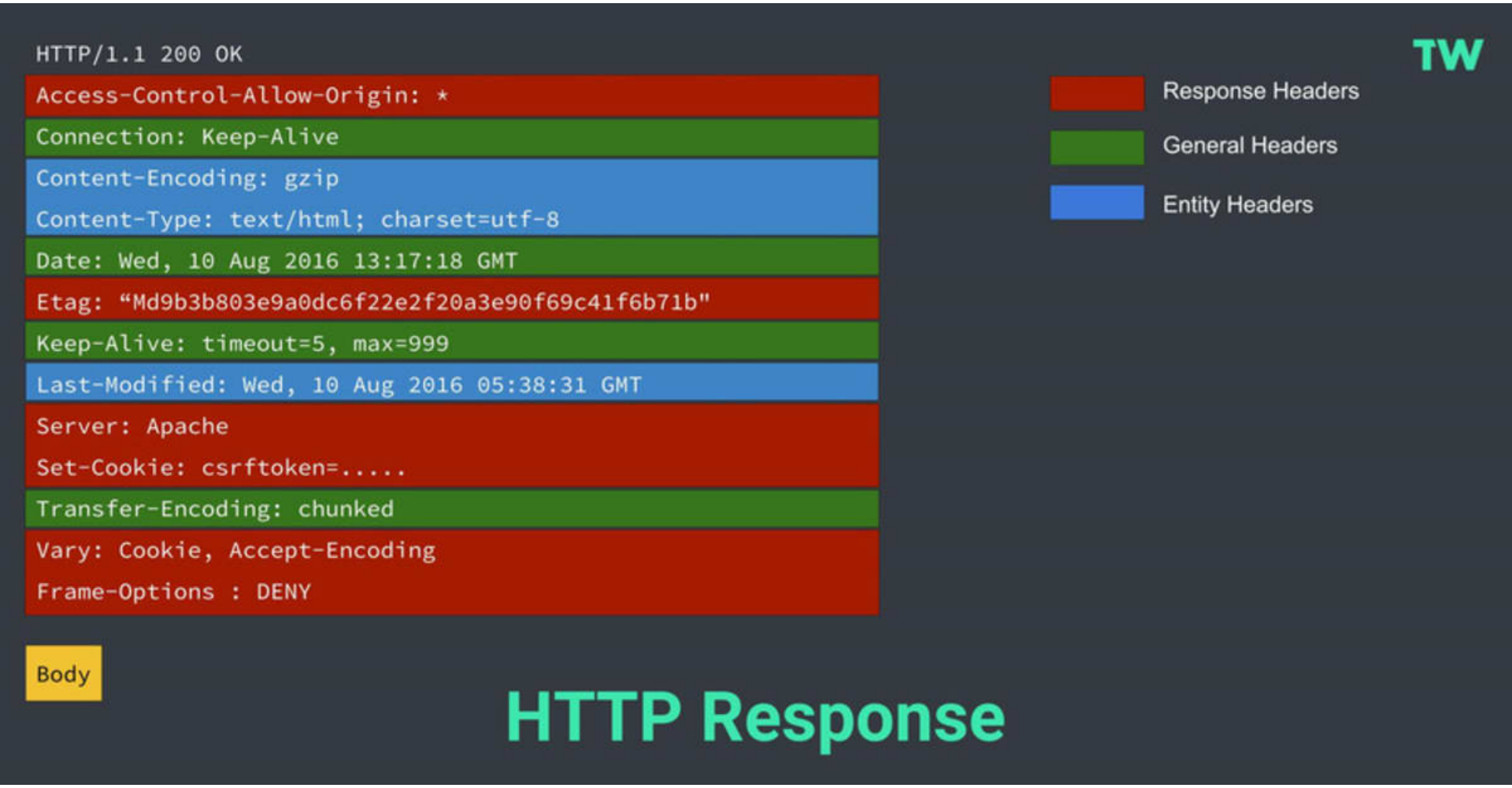
- It is binary, instead of textual
- It is fully multiplexed, instead of ordered and blocking
- It can use one connection for parallelism
- It uses header compression to reduce overhead
- It allows Server Pushing to add responses proactively into the Browser cache.



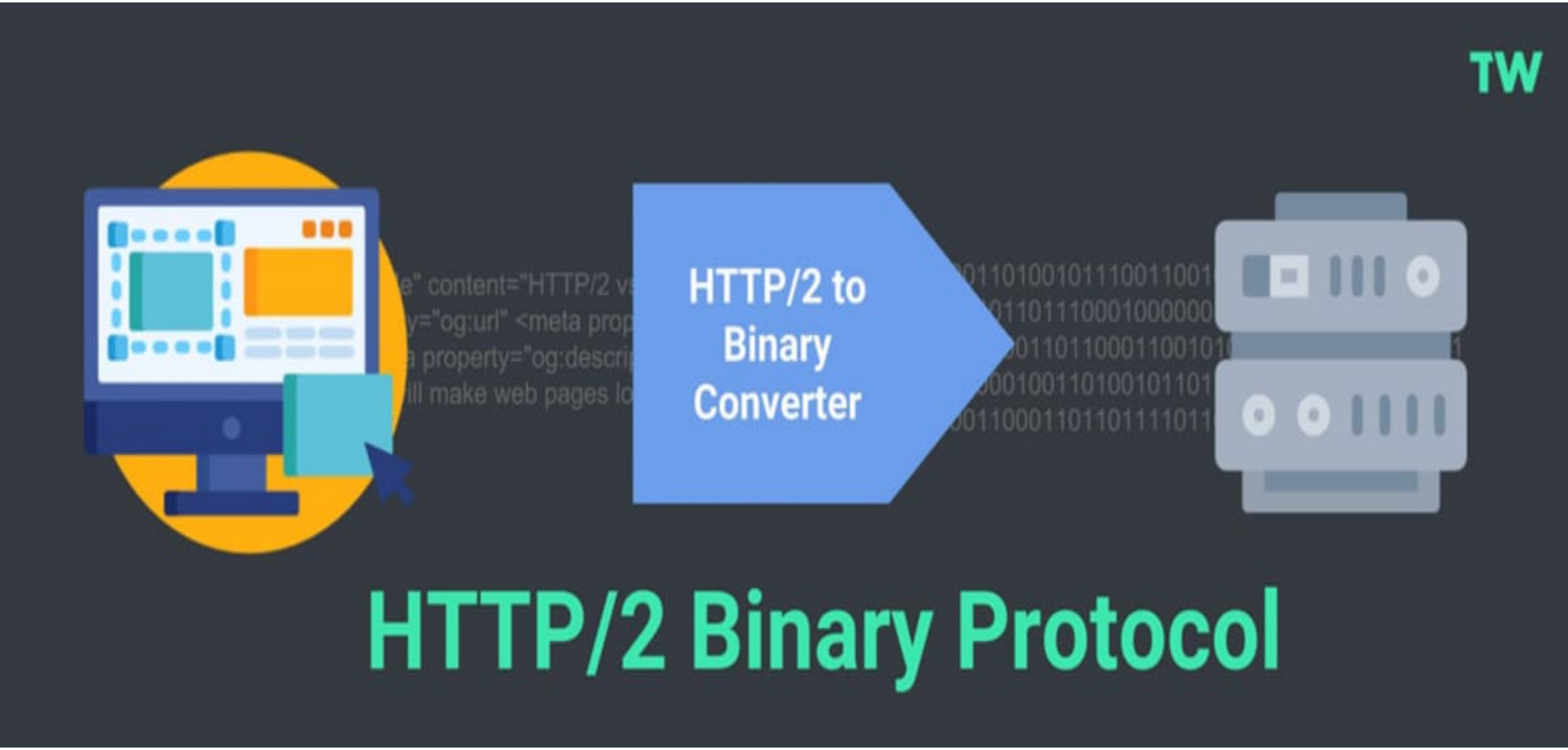
HTTP/2 vs. HTTP/1



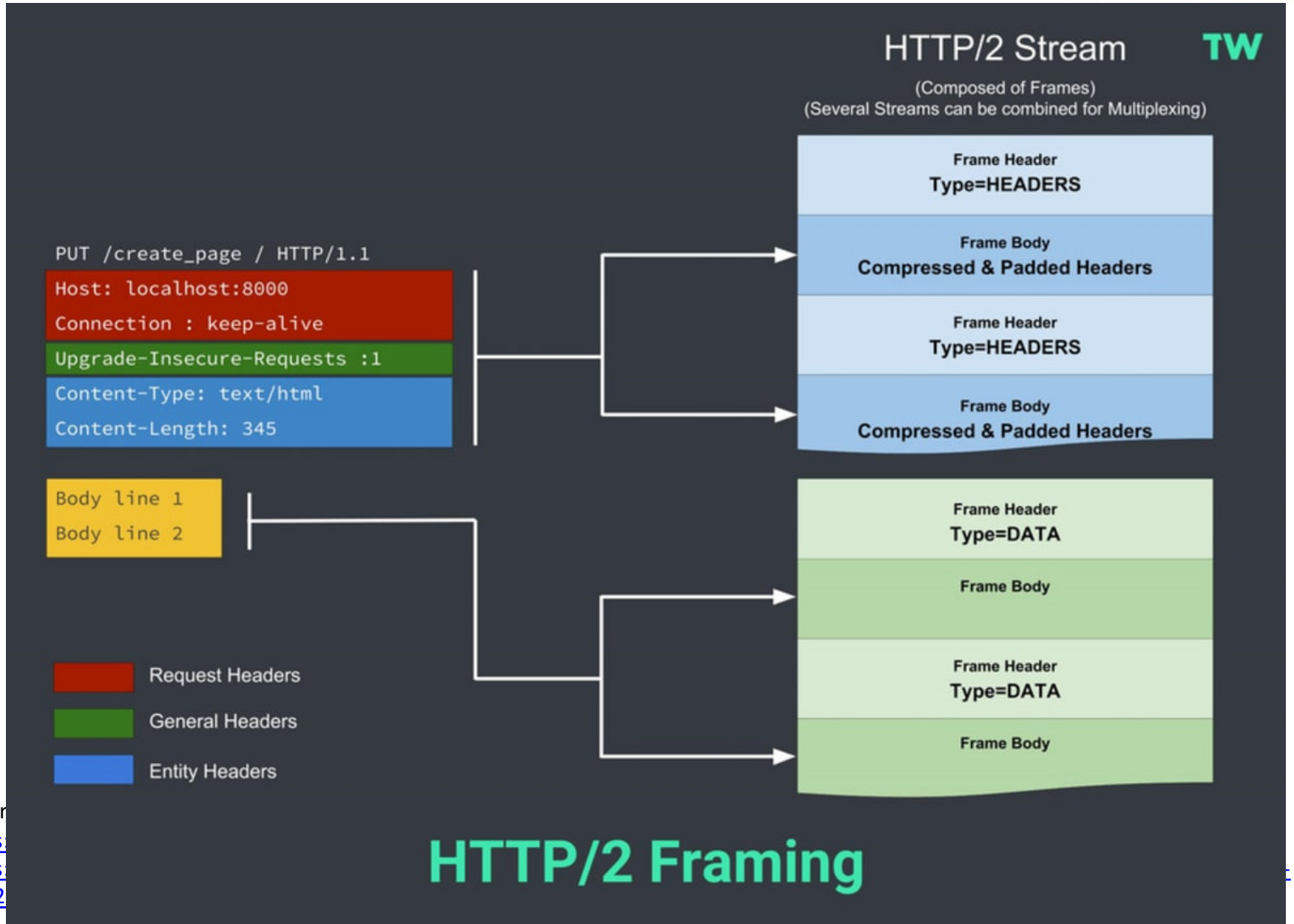
HTTP/2 vs. HTTP/1



HTTP/2 vs. HTTP/1



HTTP/2 vs. HTTP/1

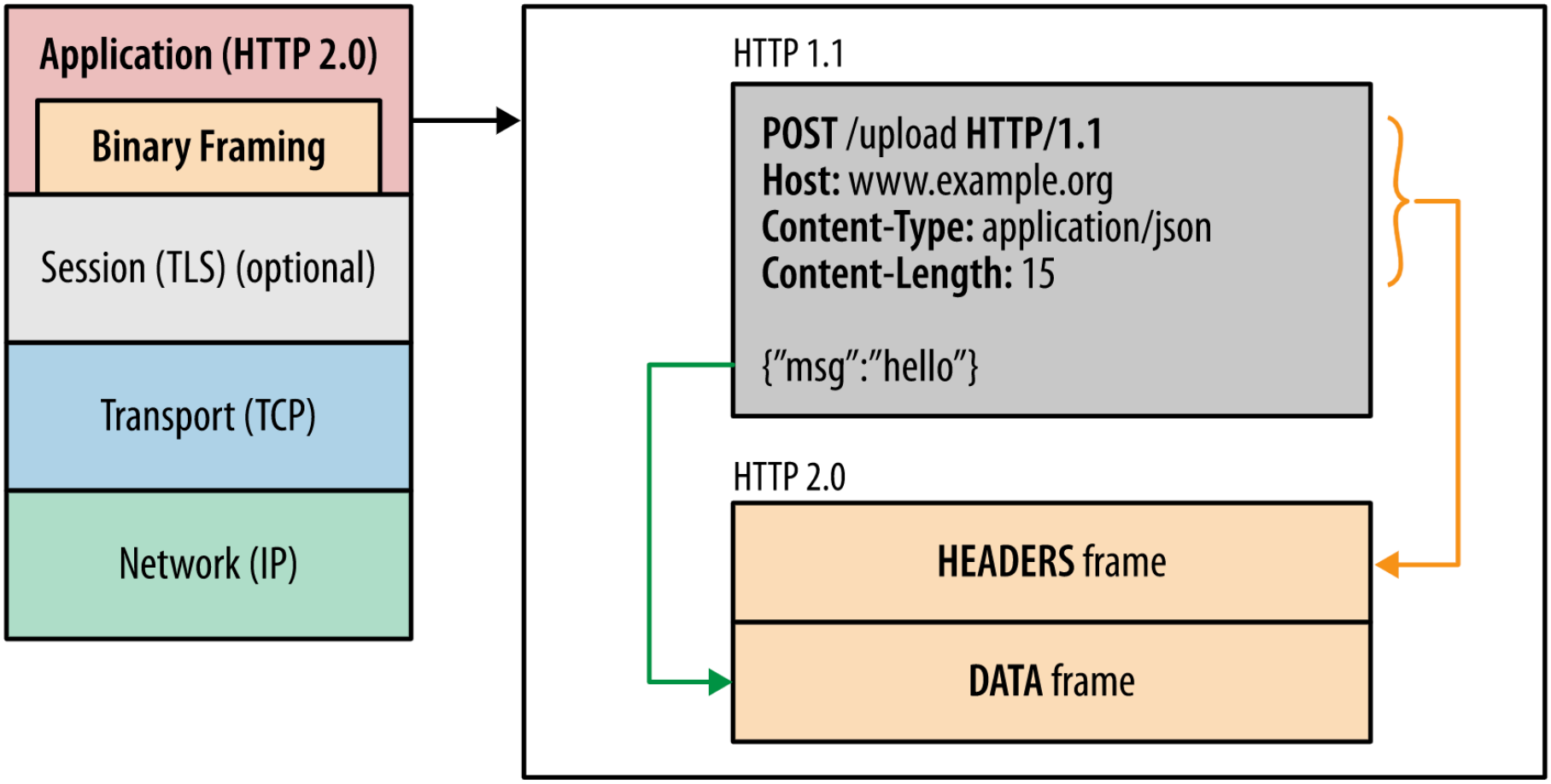


Copyr
https
https
http2

HTTP/2 vs. HTTP/1

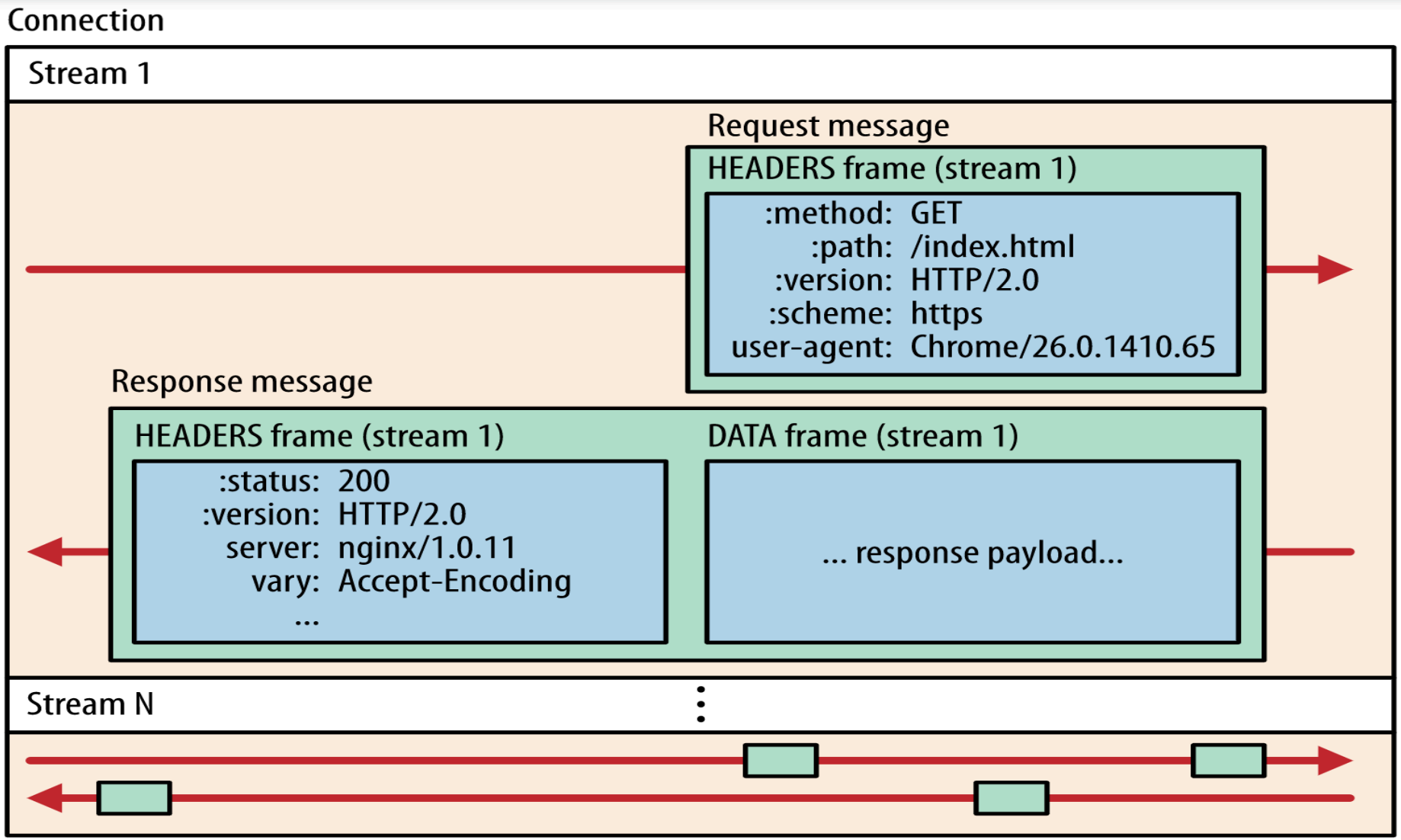
Binary framing layer

At the core of all performance enhancements of HTTP/2 is the new binary framing layer, which dictates how the HTTP messages are encapsulated and transferred between the client and server.



http2-support-on-ubuntu-16-04/ | https://developers.google.com/web/fundamentals/performance/http2

HTTP/2 vs. HTTP/1



In short, HTTP/2 breaks down the HTTP protocol communication into an exchange of binary-encoded frames, which are then mapped to messages that belong to a particular stream, all of which are multiplexed within a single TCP connection. This is the foundation that enables all other features and performance optimizations provided by the HTTP/2

Multiplexing

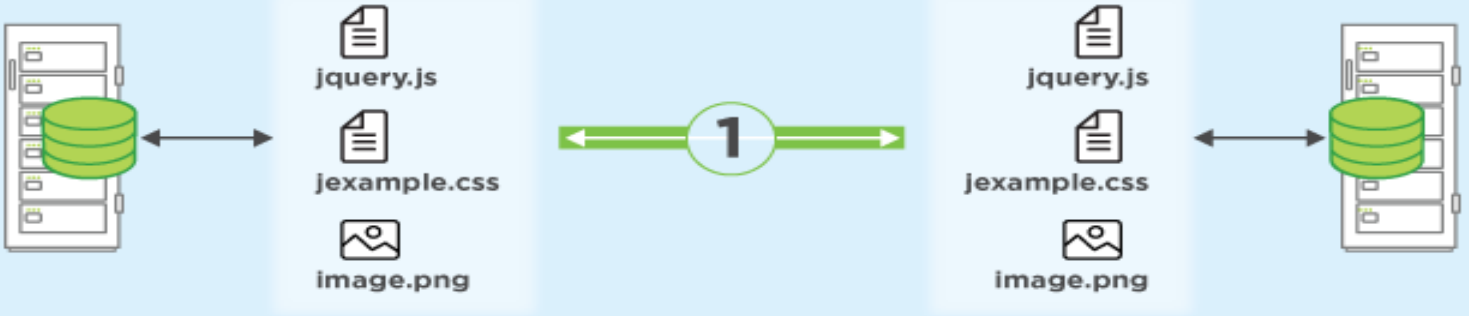
Upwork

HTTP 1.1

3 TCP CONNECTIONS

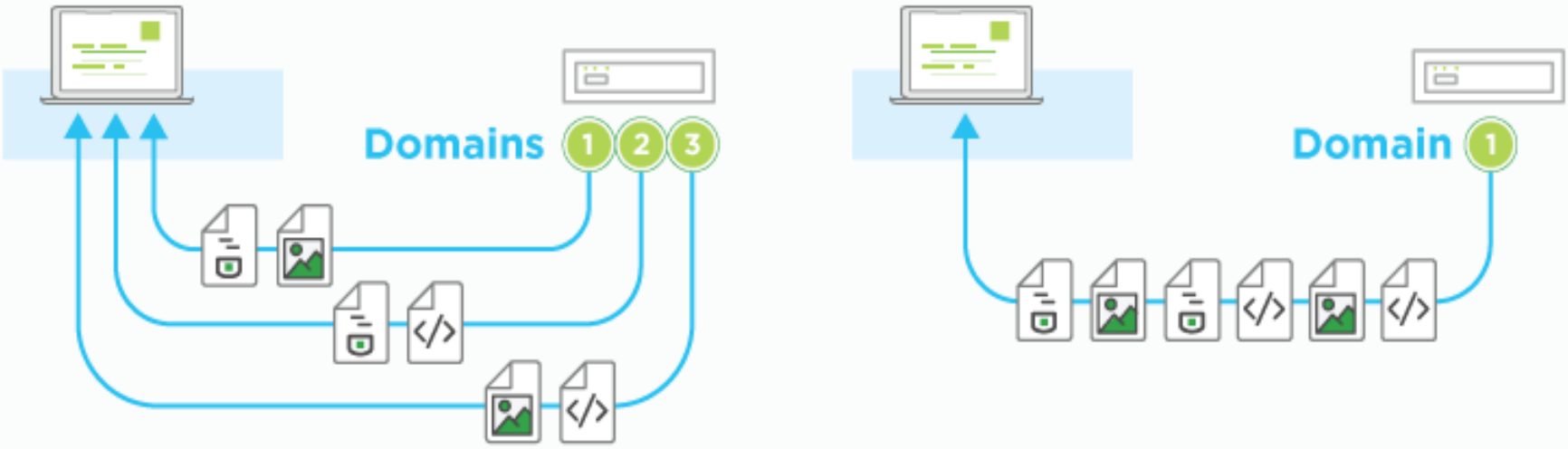
HTTP/2

1 TCP CONNECTION



HTTP/2 vs. HTTP/1

Domain Sharding



With HTTP/1.x if a user wanted to make multiple parallel requests to improve performance, they would need to use a technique such as **Domain Sharding**.
This is where a user would use a subdomain (or multiple subdomains) for assets such as images, CSS files, and JavaScript files so that they could make two or three times the number of connections to speed up the download of files.

HTTP/2 vs. HTTP/1

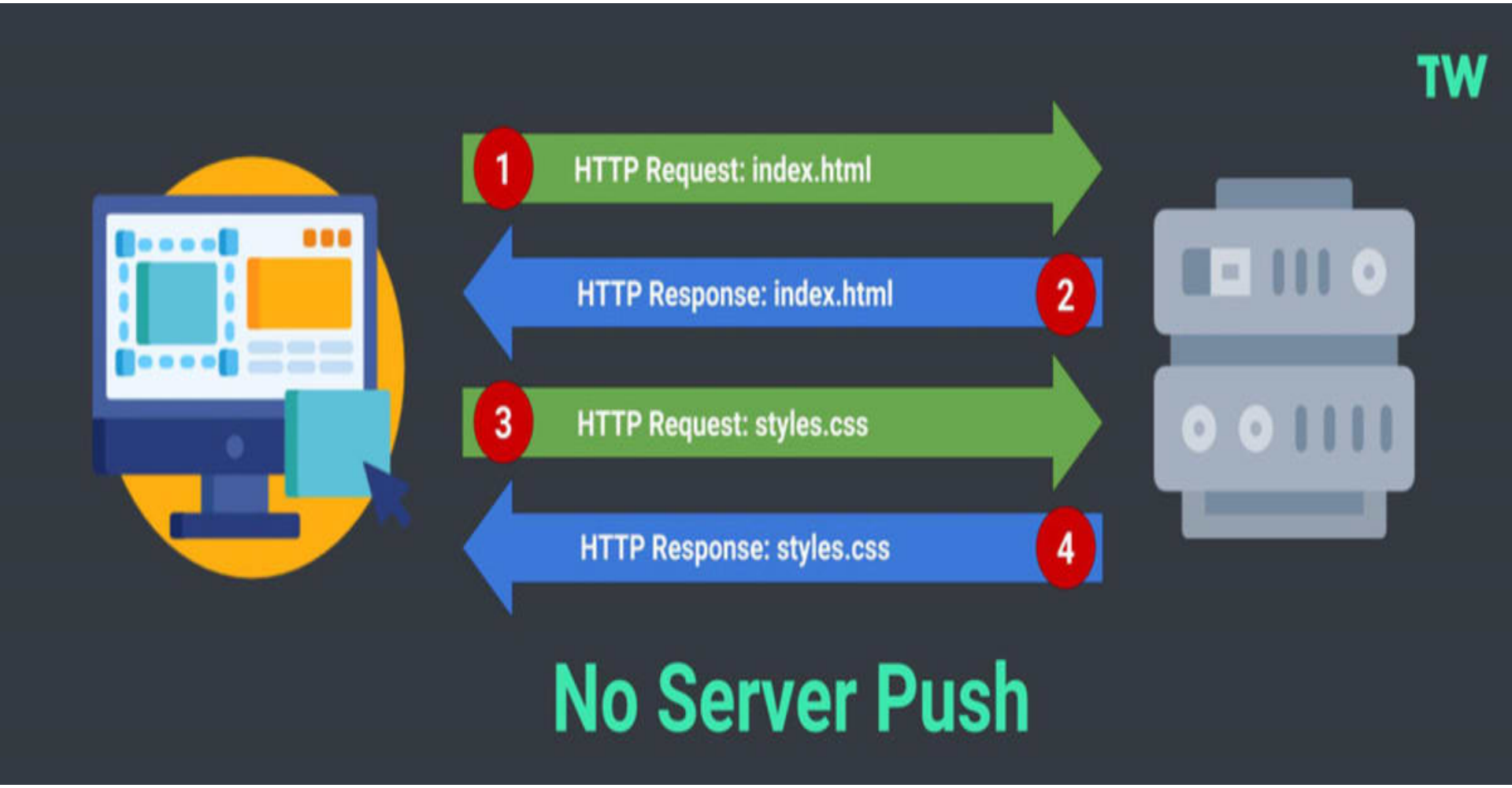
Browser Compatibility

Upwork

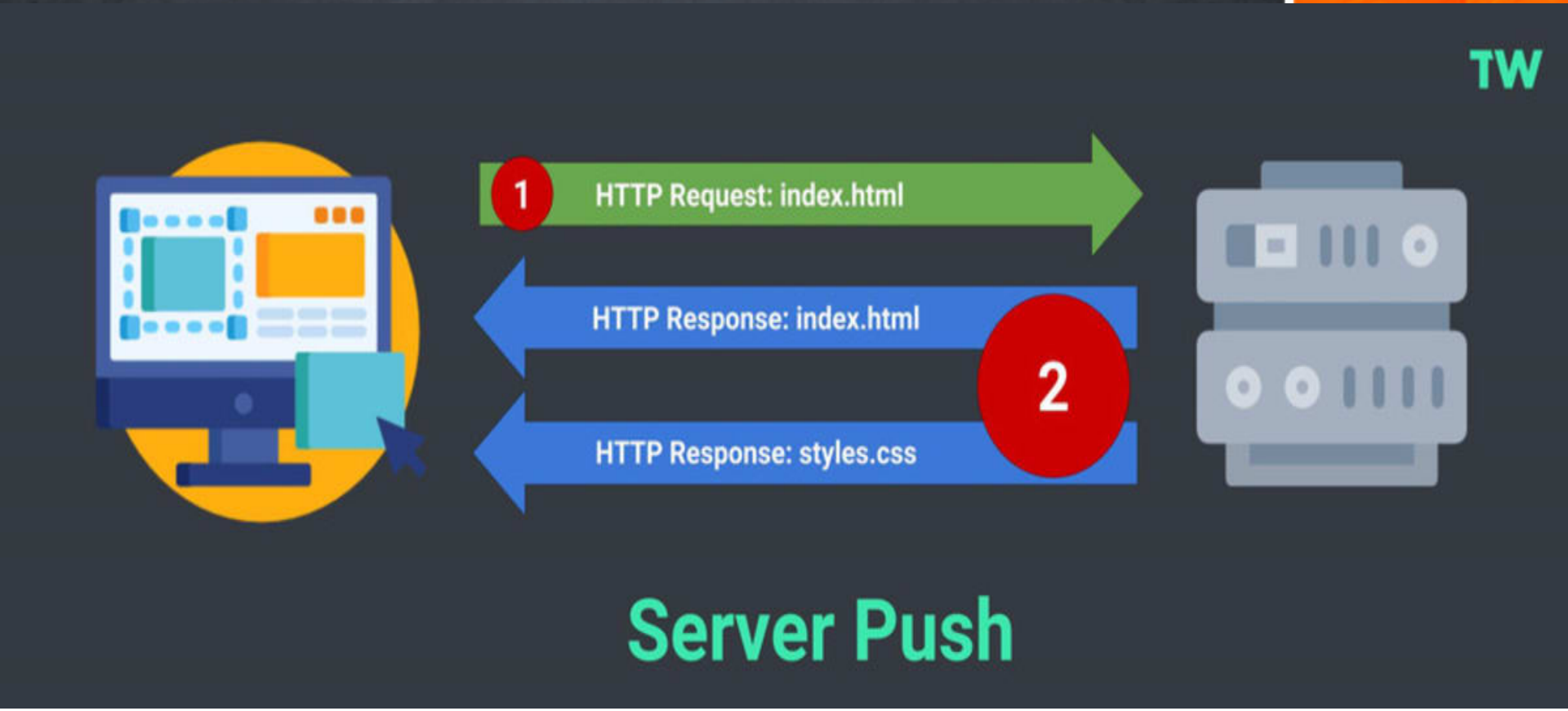
IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Chrome for Android
			29					4.3	
			49					4.4	
			50					4.4.4	
8	13	47	51			9.2			
11	14	48	52	9.1	39	9.3	all	51	51
		49	53	10	40				
		50	54	TP	41				
		51	55						

Browser HTTP/2 Support (SSL Only)

HTTP/2 vs. HTTP/1

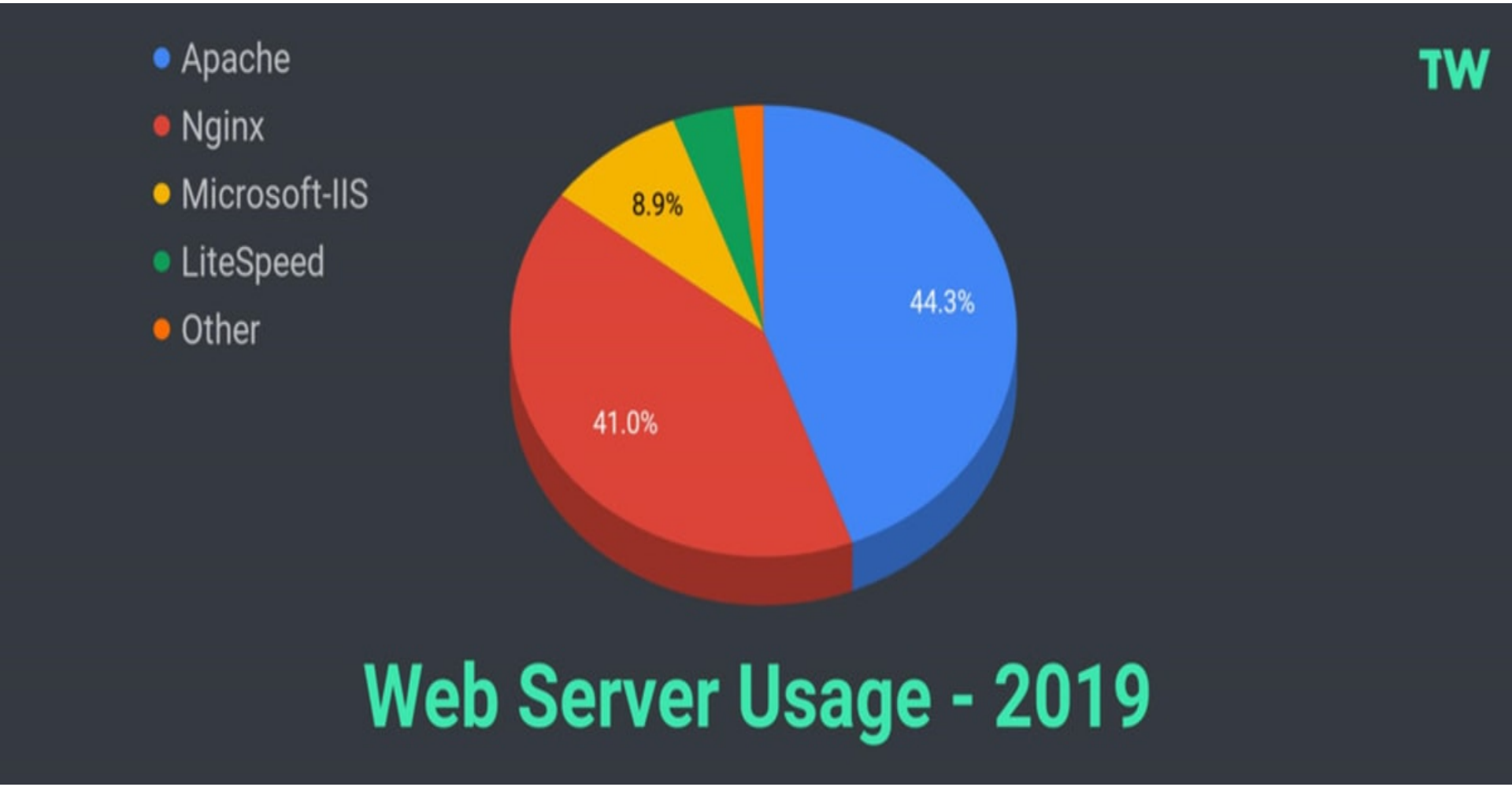


HTTP/2 vs. HTTP/1



Java Servlet PUSH example:
https://javax0.wordpress.com/2018/07/25/http-2-server-push/

HTTP/2 vs. HTTP/1



Section Conclusion

Fact: **DAD core is based on REST**

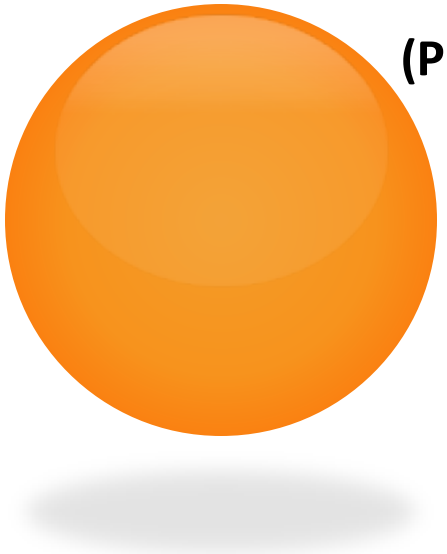
In few **samples** it is simple to remember: REST vs. SOA-WS

Parallel Computing in Distributed Systems vs. Parallel Systems

OpenMPI (HTC+HPC) vs. OpenMP (HPC)

Some “myths”:

(Distributed Systems).Equals(Distributed Computing) == true?



(Parallel System).Equals(Parallel Computing) == true?

(Parallel System == Distributed System) != true?

(Sequential vs. Parallel vs. Concurrent vs. Distributed *Programming*) *? (Different) : (Same)*

if (HTC != HPC)

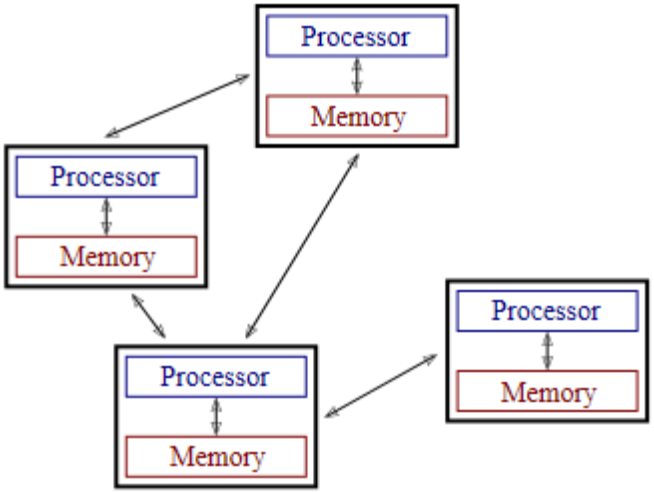
HTC (High Throughput Computing) >

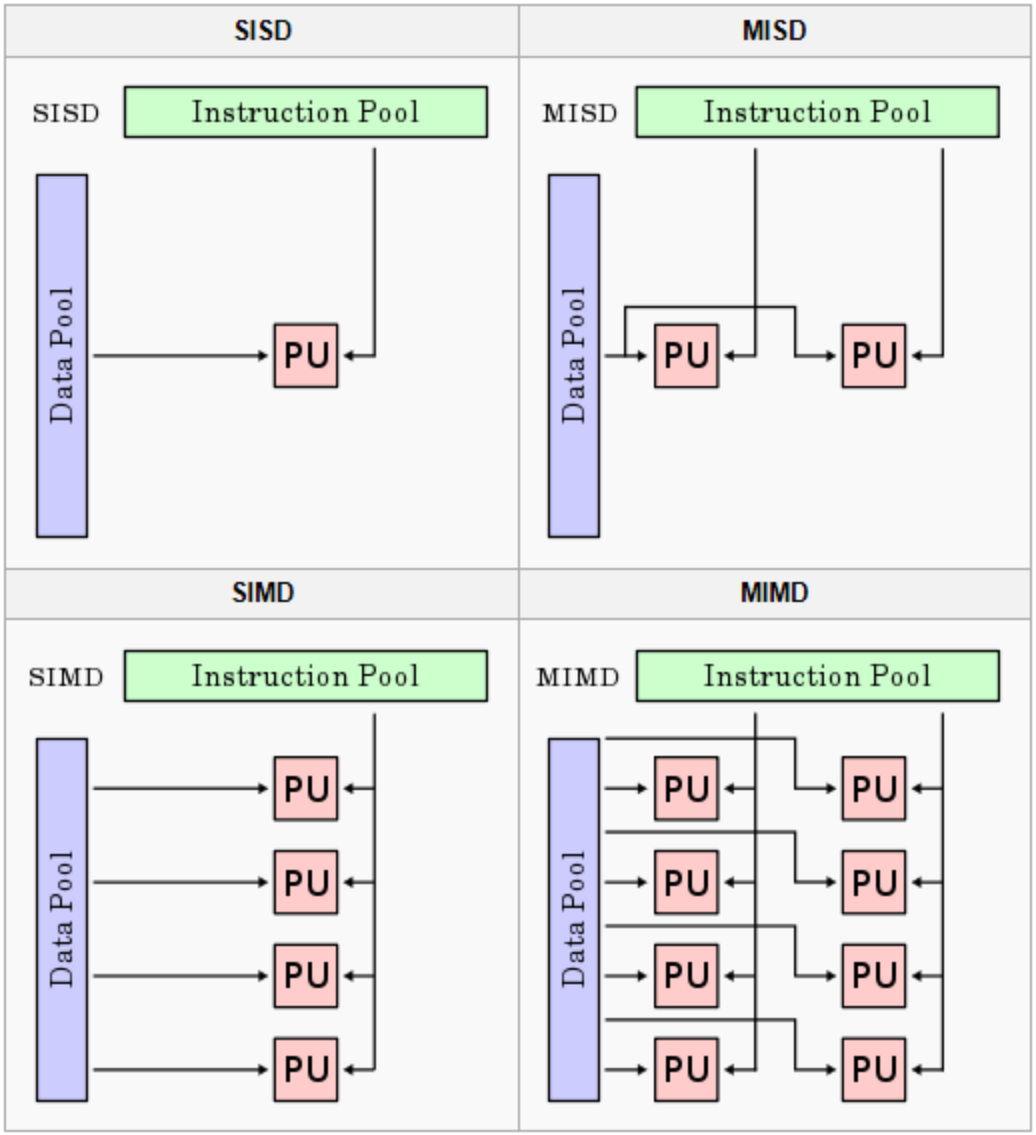
MTC (Many Task Computing) >

HPC (High Performance Computing);

Flynn Taxonomy
Parallel vs. Distributed Systems

Parallel vs. Distributed Computing / Algorithms





Where is the picture for:
Distributed System and ***Parallel System***?

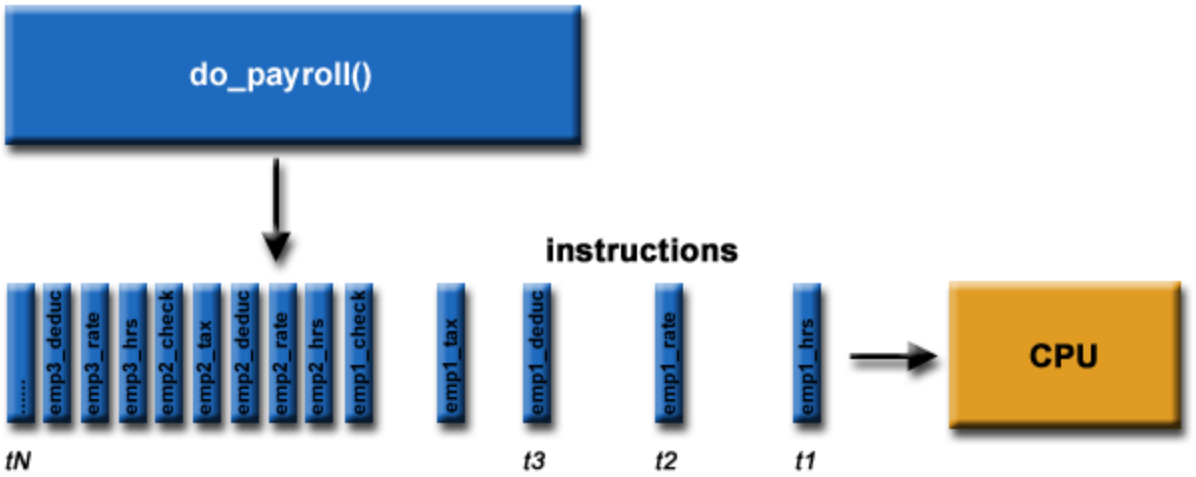
http://en.wikipedia.org/wiki/Distributed_computing

http://en.wikipedia.org/wiki/Flynn's_taxonomy

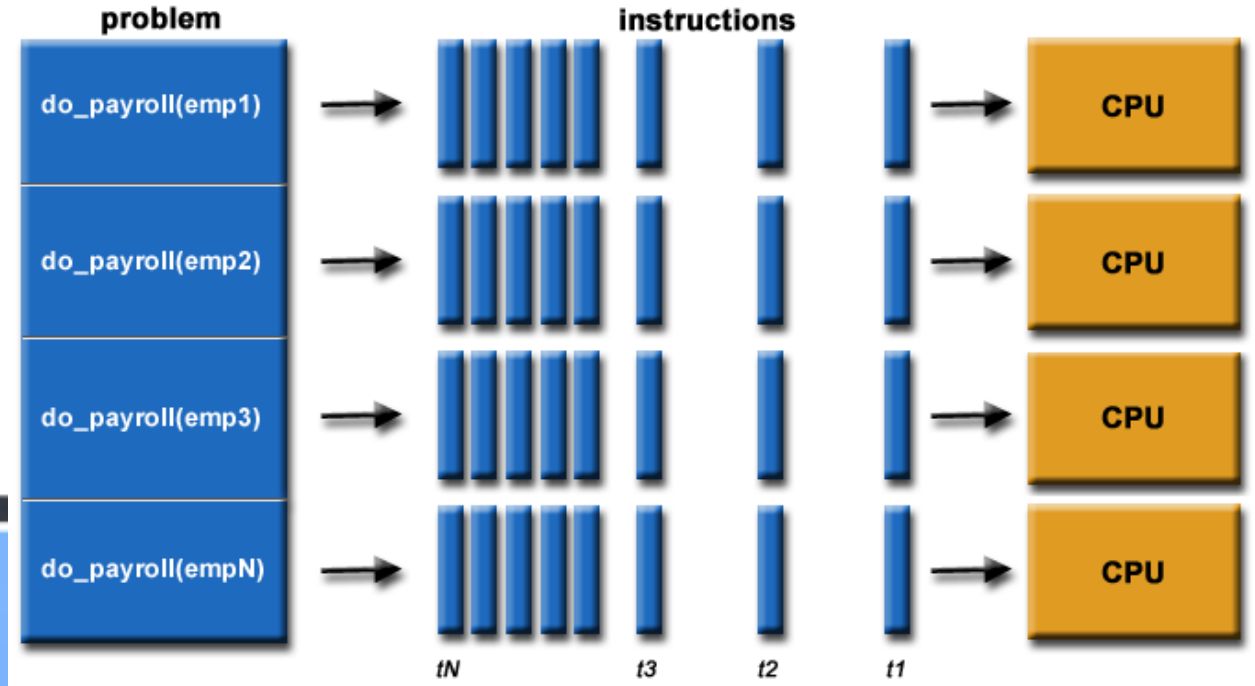
Parallel Computing & Systems - Intro

https://computing.llnl.gov/tutorials/parallel_comp/

Serial Computing



Parallel Computing



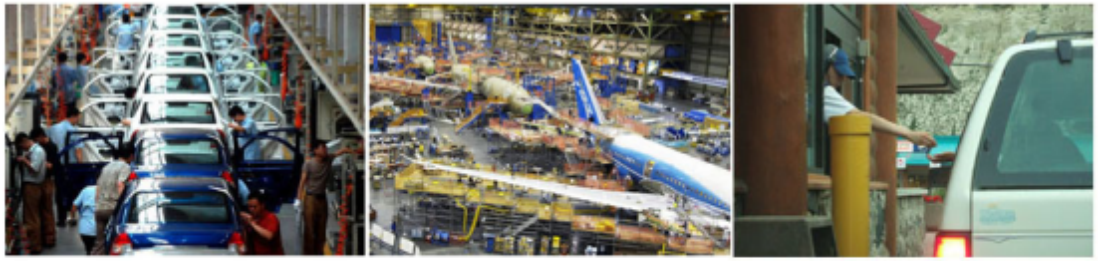
Parallel Computing & Systems - Intro

https://computing.llnl.gov/tutorials/parallel_comp/

The Real World is Massively Parallel

Galaxy Formation | Planetary Movments | Climate Change

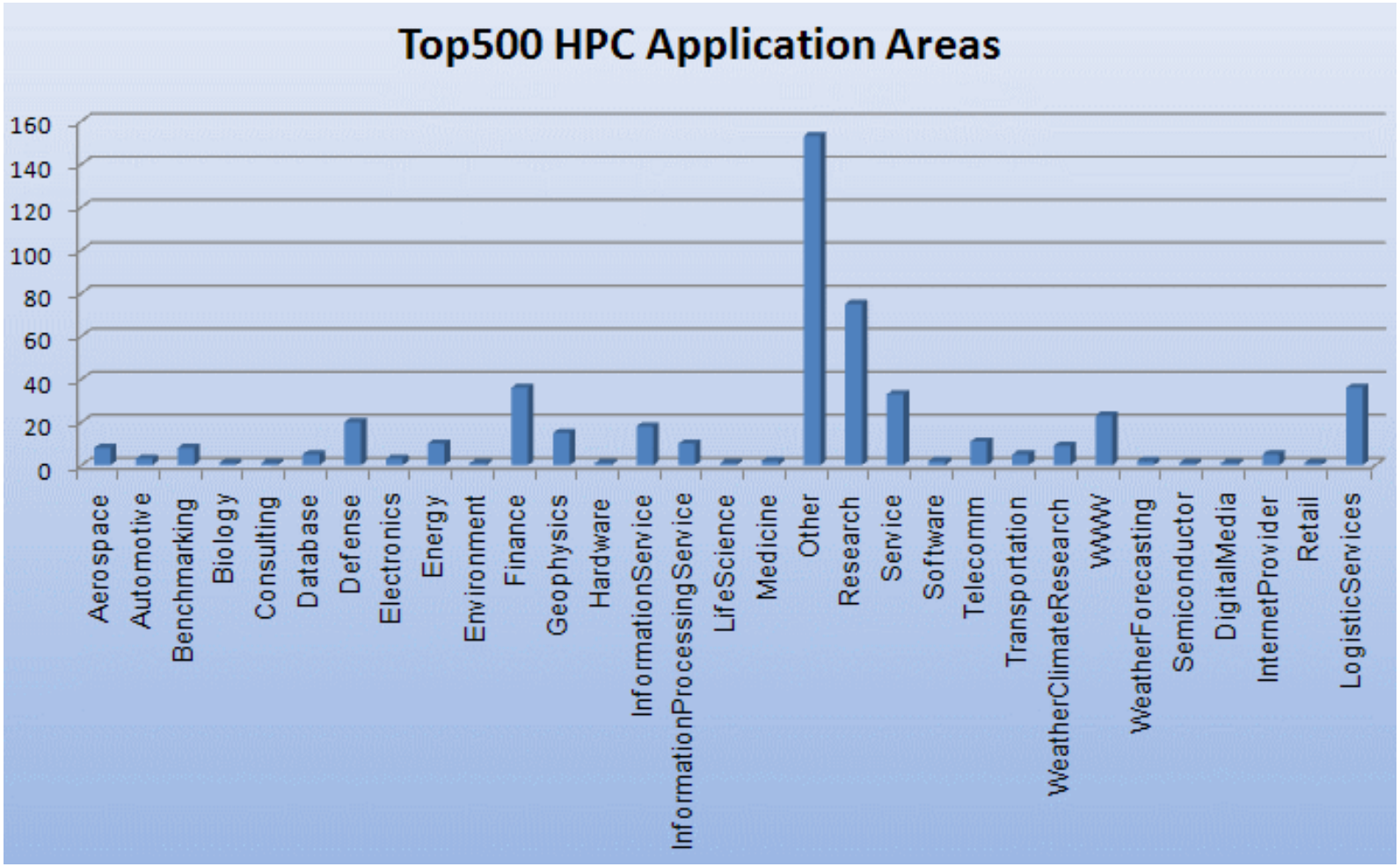
Rush Hour Traffic | Plate Tectonics | Weather



Auto Assembly | Jet Construction | Drive-thru Lunch

Parallel Computing & Systems - Intro

https://computing.llnl.gov/tutorials/parallel_comp/



Intel Accepted Technologies Overview

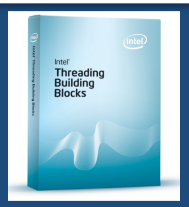
Parallel Programming

C/C++ in Linux with:

MP – Multi-processing Programming (OpenMP)	MPI – Message Passing Interface (OpenMPI)	TBB – Thread Building Blocks (Intel TBB)	OpenCL – Open Computing Language (Intel OpenCL SDK)	Multi-threaded Parallel Computing (Intel Cilk Plus, POSIX Threads, C++'11 Multithread)

OpenMP

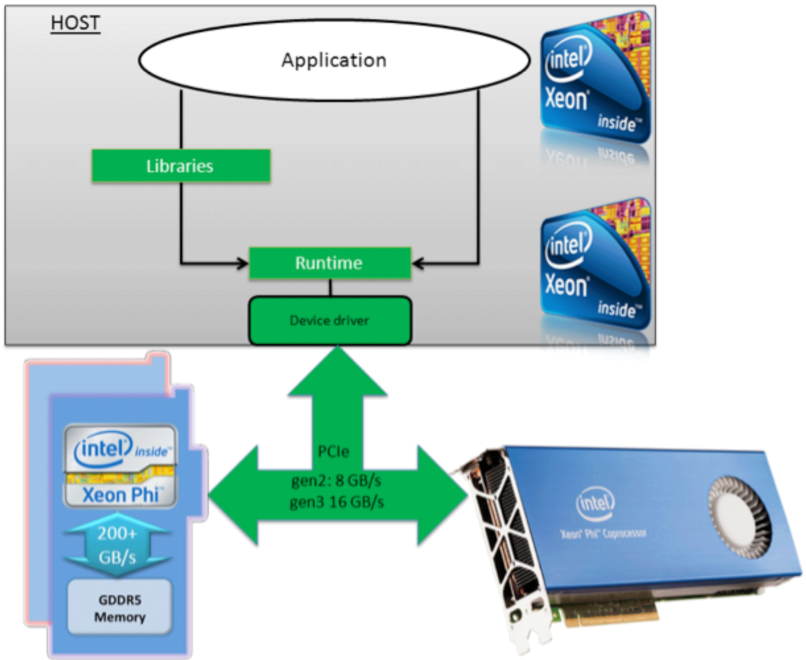
Open MPI: Open Source HPC

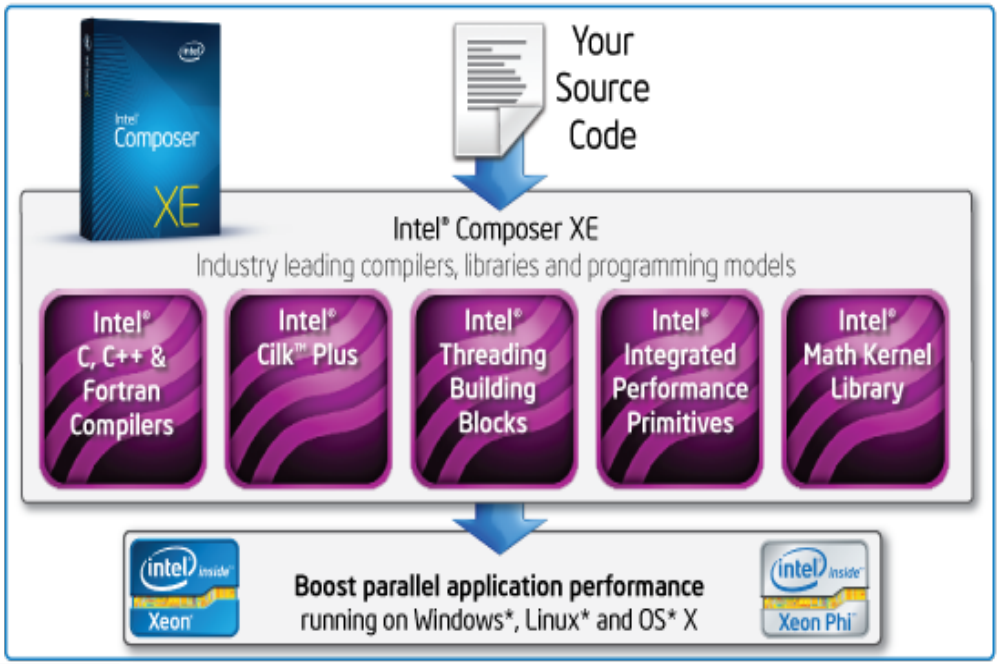


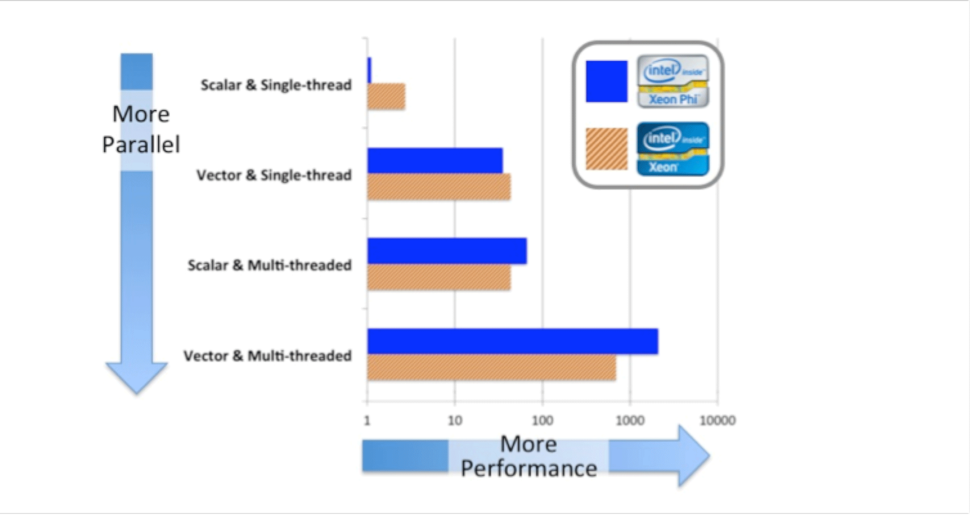
OpenCL

Intel® Cilk™ Plus
C/C++ compiler extension for simplified parallelism

HW & SW Platform







Alternative to:

1. C/C++ Nvidia CUDA

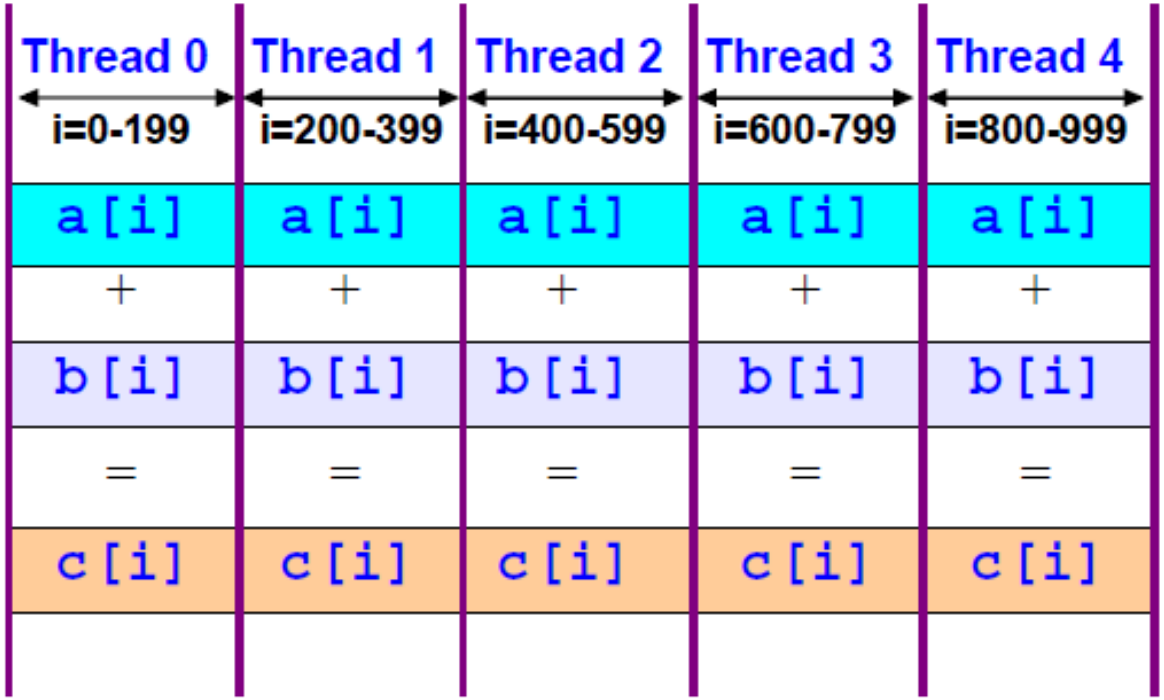
2. C/C++ OpenCL

programming on GPU – video boards

Vector Adding with Parallel Computing

http://ism.ase.ro/ | http://acs.ase.ro/java

Download the VM-Ware virtual machine with Linux Ubuntu x64 - Intel 2 cores, RAM 6048MB, HDD 25 GB



Adding two vectors sample:

- POSIX Threads
- C++'11 Threads
- Java Threads
- C++ (in OpenMP) – OpenMP mini Tutorial

Parallel Programming Restrictions

Hardware : We use a variety of servers and Xeon PHI accelerators (60 cores, 240 threads). Your code has to be portable between the execution environments, but the large workloads will be run on the Xeon PHI only.
Xeon PHI has its own small operating system and libraries, all you can use is : intel compiler, C/C++, OpenMP, TBB, MPI, Cilk (also available on servers). We only accept native Xeon PHI code for this contest, no offloading. *Late edit : we discourage you from using OpenCL, it's not optimal for this problem and without offloading.*

http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

Parallel Programming Restrictions

http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

The source code for C/C++ can be compiled without modification by the Intel compiler (icc), to run in the following modes:

- ***Native:*** The entire application runs on the Intel Xeon Phi.
- ***Offload:*** The host processor runs the application and offloads compute intensive code and associated data to the device as specified by the programmer via pragmas in the source code.
- ***Host:*** Run the code as a traditional OpenMP application on the host.

```
# compile for host-based OpenMP
icc -mkl -O3 -no-offload -openmp -Wno-unknown-pragmas -std=c99 -vec-report3
\
   matrix.c -o matrix.omp
# compile for offload mode
icc -mkl -O3 -offload-build -Wno-unknown-pragmas -std=c99 -vec-report3 \
matrix.c -o matrix.off
# compile to run natively on the Xeon Phi
icc -mkl -O3 -mmic -openmp -L /opt/intel/lib/mic -Wno-unknown-pragmas \
-std=c99 -vec-report3 matrix.c -o matrix.mic -liomp5
```

```
1  /* matrix.c (Rob Farber) */
2  #ifndef MIC_DEV
3  #define MIC_DEV 0
4  #endif
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <omp.h>
9  #include <mkl.h>
10 #include <math.h>
11
12 // An OpenMP simple matrix multiply
13 void doMult(int size, float (* restrict A)[size],
14         float (* restrict B)[size], float (* restrict C)[size])
15 {
16 #pragma offload target(mic:MIC_DEV) \
17                 in(A:length(size*size)) in( B:length(size*size))    \
18                 out(C:length(size*size))
19   {
20     // Zero the C matrix
21 #pragma omp parallel for default(none) shared(C,size)
22     for (int i = 0; i < size; ++i)
23       for (int j = 0; j < size; ++j)
24         C[i][j] =0.f;
25
26     // Compute matrix multiplication.
27 #pragma omp parallel for default(none) shared(A,B,C,size)
28     for (int i = 0; i < size; ++i)
29       for (int k = 0; k < size; ++k)
30         for (int j = 0; j < size; ++j)
31           C[i][j] += A[i][k] * B[k][j];
32   }
33 }
```

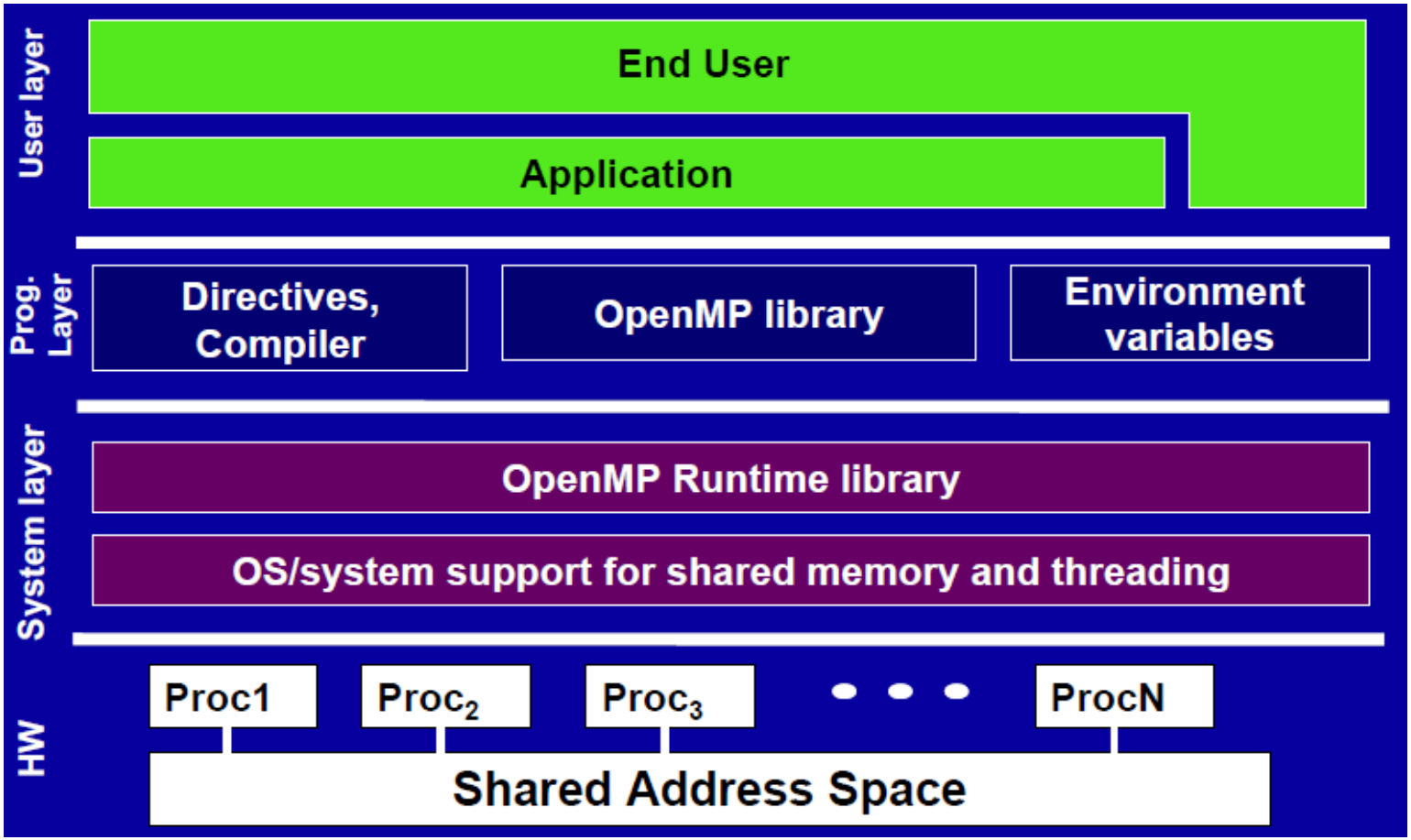
Open MP

Open specifications for Multi Processing

Long version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

- An Application Program Interface (API) that is used to explicitly direct multi-threaded, shared memory parallelism.
- API components:
  - Compiler directives (Compilers that supports: GNU & Intel C/C++ compilers - **gcc/g++** & **icc**)
  - Runtime library routines
  - Environment variables
- Portability
  - API is specified for C/C++ and Fortran
  - Implementations on almost all platforms including Unix/Linux and Windows
- Standardization
  - Jointly defined and endorsed by major computer hardware and software vendors
  - Possibility to become ANSI standard

OpenMP Architecture – Version 3



OpenMP Mini-Tutorial – Version 3

Thread

- A **process** is an instance of a computer program that is being executed. It contains the program code and its current activity.
- A **thread of execution** is the smallest unit of processing that can be scheduled by an operating system.
- Differences between threads and processes:
  - A thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory. The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment).
  - Different processes do not share these resources.

http://en.wikipedia.org/wiki/Process_(computing) |

Process

- A process contains all the information needed to execute the program:
  - Process ID
  - Program code
  - Data on run time stack
  - Global data
  - Data on heap

Each process has its own address space.

- In multitasking, processes are given time slices in a round robin fashion.
  - If computer resources are assigned to another process, the status of the present process has to be saved, in order that the execution of the suspended process can be resumed at a later time.

OpenMP - Mini-Tutorial – Version 3

Threads Features:

- Thread model is an extension of the process model.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process share the address space of this process.
  - Global variables and all dynamically allocated data objects are accessible by all threads of a process.
- Each thread has its own run time stack, register, program counter.
- Threads can communicate by reading/writing variables in the common address space.

Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.

A thread does not maintain a list of created threads, nor does it know the thread that created it.

All threads within a process share the same address space.

Threads in the same process share:

- Process instructions
- Most data
- open files (descriptors)
- signals and signal handlers
- current working directory
- User and group id

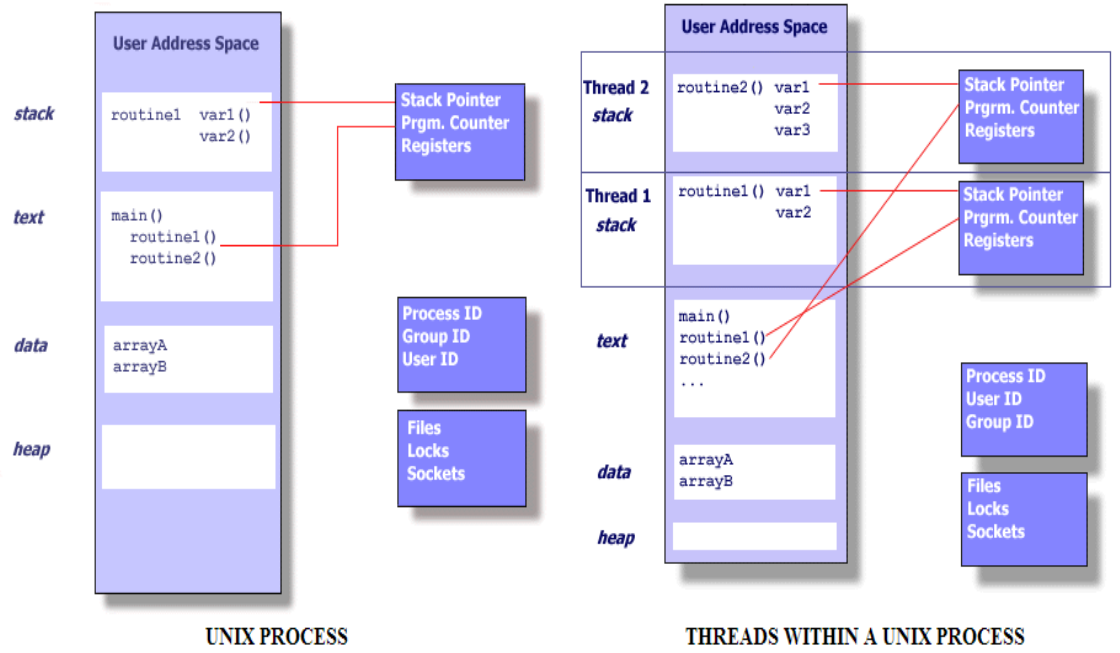
Each thread has a unique:

- Thread ID
- set of registers, stack pointer
- stack for local variables, return addresses
- signal mask
- priority
- Return value: errno

pthread functions return "0" if OK.

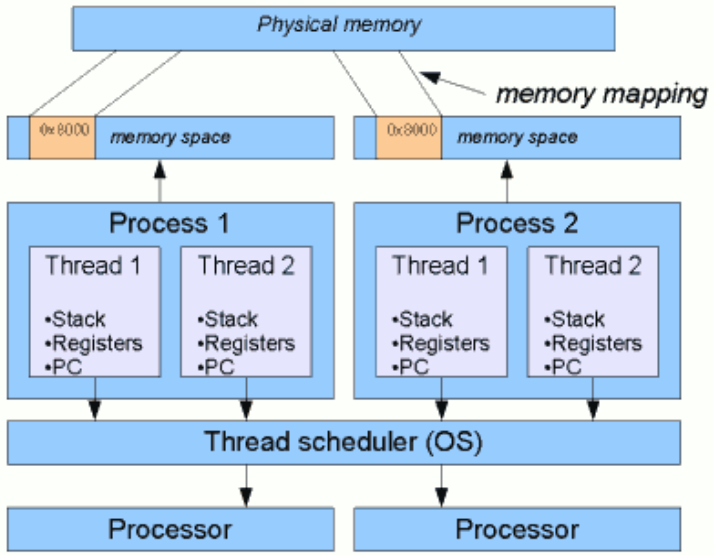
OpenMP - Mini-Tutorial – Version 3

Multi-threading vs. Multi-process development in UNIX/Linux:



UNIX PROCESS

THREADS WITHIN A UNIX PROCESS



https://computing.llnl.gov/tutorials/pthreads/

http://www.javamex.com/tutorials/threads/how_threads_work.shtml

OpenMP - Mini-Tutorial – Version 3

OpenMP Programming Model:

- Shared memory, thread-based parallelism
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.
- Explicit Parallelism
  - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- Compiler directive based
  - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code.

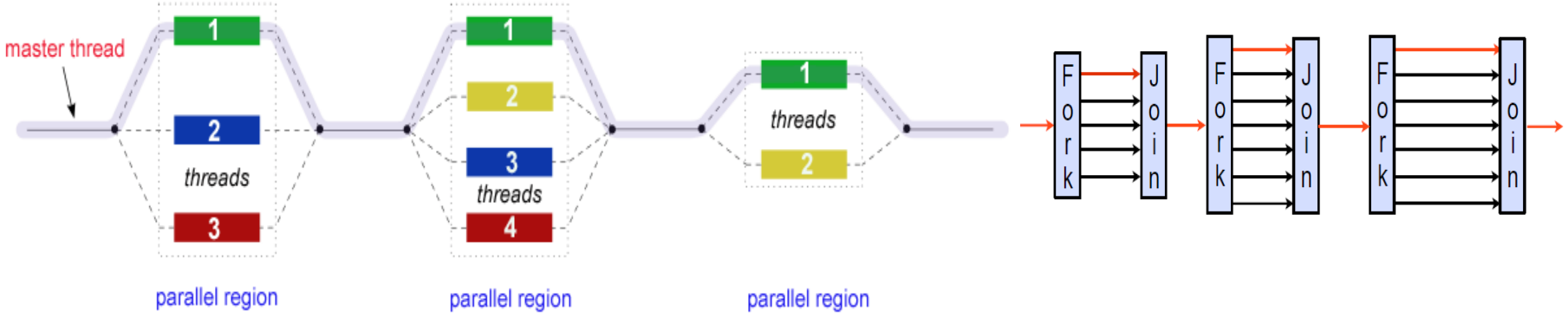
OpenMP is NOT:

- Necessarily implemented identically by all vendors
- Meant for distributed-memory parallel systems (it is designed for shared address spaced machines)
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

OpenMP - Mini-Tutorial – Version 3

OpenMP - Fork-Join Parallelism Model:

- OpenMP program begin as a single process: the *master thread (in pictures in red/grey)*. The master thread executes sequentially until the first *parallel region* construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by **FORK**.
  - Becomes the master of this group of threads, and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the *parallel region* construct are then executed in parallel among these threads.
- **JOIN**: When the threads complete executing the statement in the *parallel region* construct, they synchronize and terminate, leaving only the master thread.



OpenMP - Mini-Tutorial – Version 3

I/O

- OpenMP does not specify parallel I/O.
- It is up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Memory Model

- Threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
- When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is updated by all threads as needed.

```
//OpenMP Code Structure
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main()
{
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("Hello (%d)\n", ID);
    printf(" world (%d)\n", ID);
  }
}
```

Set # of threads for OpenMP:

- In csh:

```
setenv OMP_NUM_THREAD 8
```

- In bash:

```
set OMP_NUM_THREAD=8
export $OMP_NUM_THREAD
```

Compile: g++ -fopenmp hello.c

Run: ./a.out

Compiler / Platform	Compiler	Flag
Intel Linux Opteron/Xeon	icc icpc ifort	-openmp
PGI Linux Opteron/Xeon	pgcc pgCC pgf77 pgf90	-mp
GNU Linux Opteron/Xeon IBM Blue Gene	gcc g++ g77 gfortran	-fopenmp

OpenMP - Mini-Tutorial – Version 3

OpenMP Core Syntax

```
#include “omp.h”
void main ()
{
      int var1, var2, var3;
      // 1. Serial code
      . . .
      // 2. Beginning of parallel section.
      // Fork a team of threads. Specify variable scoping
      #pragma omp parallel private(var1, var2) shared(var3)
      {
      // 3. Parallel section executed by all threads
      . . .
      // 4. All threads join master thread and disband
      }
      // 5. Resume serial code . . .
}
```

OpenMP C/C++ Directive Format

- OpenMP directive forms
  - C/C++ use compiler directives
- Prefix: #pragma omp …
  - A directive consists of a directive name followed by *clauses*

Example:

```
#pragma omp parallel default (shared)
private (var1, var2)
```

OpenMP Directive Format - General Rules:

- Case sensitive
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash “\” at the end of a directive line.

OpenMP - Mini-Tutorial – Version 3

OpenMP *parallel* Region Directive

#pragma omp parallel [*clause list*]

Typical clauses in [*clause list*]

- Conditional parallelization
  - if (scalar expression)
    - Determine whether the parallel construct creates threads
- Degree of concurrency
  - num_threads (integer expression)
    - number of threads to create
- Date Scoping
  - private (variable list)
    - Specifies variables local to each thread
  - firstprivate (variable list)
    - Similar to the private
    - Private variables are initialized to variable value before the parallel directive
  - shared (variable list)
    - Specifies variables that are shared among all the threads
  - default (data scoping specifier)
    - Default data scoping specifier may be shared or none

Example:

```
#pragma omp parallel if (is_parallel == 1)
num_threads(8) shared (var_b) private (var_a)
firstprivate (var_c) default (none)
{
/* structured block */
}
```

- if (is_parallel == 1) num_threads(8)
  - If the value of the variable is_parallel is one, create 8 threads
- shared (var_b)
  - Each thread shares a single copy of variable var_b
- private (var_a) firstprivate (var_c)
  - Each thread gets private copies of variable var_a and var_c
  - Each private copy of var_c is initialized with the value of var_c in main thread when the parallel directive is encountered
- default (none)
  - Default state of a variable is specified as none (rather than shared)
  - Signals error if not all variables are specified as shared or private

Number of Threads:

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the if clause
  2. Setting of the num_threads() clause
  3. Use of the omp_set_num_threads() library function
  4. Setting of the OMP_NUM_THREAD environment variable
  5. Implementation default – usually the number of cores on a node

- Threads are numbered from 0 (master thread) to N-1

OpenMP - Mini-Tutorial – Version 3

Thread Creation: Parallel Region Example - **Create threads with the parallel construct**

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main()
{
   int nthreads, tid;
   #pragma omp parallel num_threads(4) private(tid)
   {
      tid = omp_get_thread_num();
      printf("Hello world from (%d)\n", tid);
      if(tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("number of threads = %d\n", nthreads);
      }
   } // all threads join master thread and terminates
}
```

Clause to request threads

Each thread executes a copy of the code within the structured block

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main(){
   int nthreads,  A[100] , tid;
   // fork a group of threads with each thread having a private tid variable
   omp_set_num_threads(4);
   #pragma omp parallel private (tid)
   {
      tid = omp_get_thread_num();
      foo(tid, A);
   } // all threads join master thread and terminates
}
```

A single copy of A[] is shared between all threads

Work-Sharing Construct:

- A parallel construct by itself creates a "Single Program/Instruction Multiple Data" (SIMD) program, i.e., each thread executes the same code.
- Work-sharing is to split up pathways through the code between threads within a team.
  - Loop construct (for/do)
  - Sections/section constructs
  - Single construct
- Within the scope of a parallel directive, work-sharing directives allow concurrency between iterations or tasks
- ***Work-sharing constructs do not create new threads.***
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- ***Work-sharing constructs must be encountered by all members of a team or none at all.***
- Two directives to be presented
  - – do/for: concurrent loop iterations
  - – sections: concurrent tasks

OpenMP - Mini-Tutorial – Version 3

Work-Sharing do/for Directive

do/for:

- Shares iterations of a loop across the group
- *Represents a "data parallelism"*.

for directive partitions parallel iterations across threads
do is the analogous directive in Fortran

- Usage:

#pragma omp for [clause list]
/* for loop */

- Implicit barrier at end of for loop

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
void main()
{
  int nthreads, tid;

  omp_set_num_threads(3);

  #pragma omp parallel private(tid)
  {
     int i;
     tid = omp_get_thread_num();
     printf("Hello world from (%d)\n", tid);
     #pragma omp for
     for(i = 0; i <=4; i++)
     {
       printf("Iteration %d by %d\n", i, tid);
     }
  } // all threads join master thread and terminates
}
```

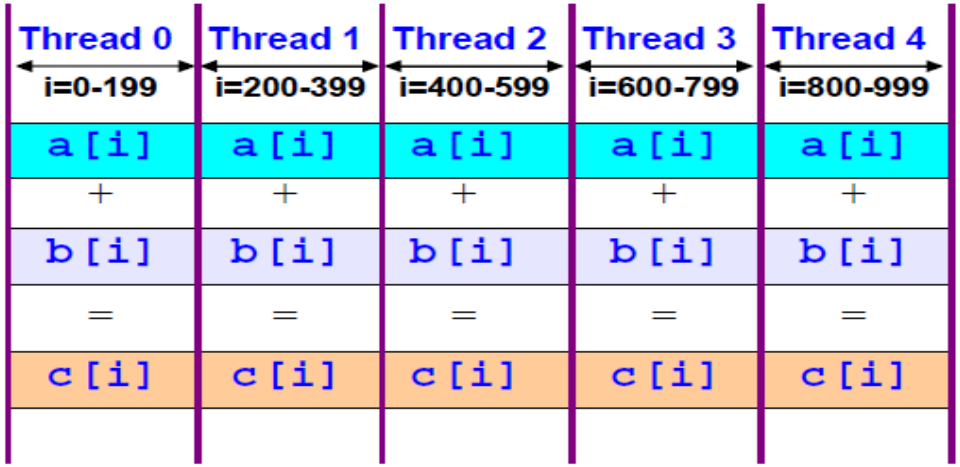
OpenMP - Mini-Tutorial – Version 3

```
//Sequential code to add two vectors:
for(i=0;i<N;i++) {
  c[i] = b[i] + a[i];
}
```

```
//OpenMP implementation 1 (not desired):
#pragma omp parallel
{
   int id, i, Nthrds, istart, iend;
   id = omp_get_thread_num();
   Nthrds = omp_get_num_threads();
   istart = id*N/Nthrds;
   iend = (id+1)*N/Nthrds;

   if(id == Nthrds-1) iend = N;
   for(I = istart; i<iend; i++) {
      c[i] = b[i]+a[i];
   }
}
```

```
//A worksharing for construct to add vectors:
#pragma omp parallel
{
   #pragma omp for
   {
      for(i=0; i<N; i++) { c[i]=b[i]+a[i]; }
   }
}
```

```
//A worksharing for construct to add vectors:
#pragma omp parallel for
for(i=0; i<N; i++) { c[i]=b[i]+a[i]; }
```



OpenMP - Mini-Tutorial – Version 3

C/C++ for Directive Syntax:

```
#pragma omp for [clause list]
        schedule (type [,chunk])
        ordered
        private (variable list)
        firstprivate (variable list)
        shared (variable list)
        reduction (operator: variable list)
        collapse (n)
        nowait
/* for_loop */
```

For Directive Restrictions

For the "*for* loop" that follows the *for* directive:

- It must not have a break statement
- The loop control variable must be an integer
- The initialization expression of the "*for* loop" must be an integer assignment.
- The logical expression must be one of <,≤,>,≥
- The increment expression must have integer increments or decrements only.

How to combine values into a single accumulation variable (avg)?

//Sequential code to do average value from an array-vector:

```
{
   double avg = 0.0, A[MAX];
   int i;
   …
   for(i =0; i<MAX; i++) {
      avg += a[i];
   }

   avg /= MAX;
}
```

OpenMP - Mini-Tutorial – Version 3

Reduction Clause

- *Reduction (operator: variable list)*: specifies how to combine local copies of a variable in different threads into a single copy at the master when threads exit. Variables in *variable list* are implicitly private to threads.
- Operators used in Reduction Clause: +, *, -, &, |, ^, &&, and ||
- Usage Sample:

```
#pragma omp parallel reduction(+: sums) num_threads(4)
{
  /* compute local sums in each thread */
}
/* sums here contains sum of all local instances of sum */
```

Reduction in OpenMP for:

Inside a parallel or a work-sharing construct:

- A local copy of each list variable is made and initialized depending on *operator* (e.g. 0 for "+")
- Compiler finds standard reduction expressions containing *operator* and uses it to update the local copy.
- Local copies are reduced into a single value and combined with the original global value when returns to the master thread.

```
//A work-sharing for average value from a vector:
{
   double avg = 0.0, A[MAX];
   int i;
   …
   #pragma omp parallel for reduction (+:avg)
   for(i =0; i<MAX; i++) {avg += a[i];}

   avg /= MAX;
}
```

Operator	Initial Value
+	0
*	1
-	0
&	~0

Operator	Initial Value
|	0
^	0
&&	1
||	0

Reduction Operators/Initial-Values in C/C++ OpenMP

OpenMP - Mini-Tutorial – Version 3

Matrix-Vector Multiplication

```
for (i=0,1,2,3,4)
 i = 0
sum =    b[i=0][j]*c[j]
  a[0] = sum
 i = 1
sum =    b[i=1][j]*c[j]
  a[1] = sum
```

Thread 0,

```
for (i=5,6,7,8,9)
 i = 5
sum =    b[i=5][j]*c[j]
  a[5] = sum
 i = 6
sum =    b[i=6][j]*c[j]
  a[6] = sum
```

Thread 1, ...etc...

```
#pragma omp parallel default (none) \
shared (a, b, c, m,n) private (i,j,sum) num_threads(4)
for(i=0; i < m; i++)
{
        sum = 0.0;
        for(j=0; j < n; j++)
                sum += b[i][j]*c[j];

        a[i] =sum;
}
```

OpenMP - Mini-Tutorial – Version 3

Matrix-Vector | Matrix-Matrix Multiplication

schedule clause

- Describe how iterations of the loop are divided among the threads in the group. The default schedule is implementation dependent.
- Usage: schedule (scheduling_class[, parameter]).

– ***static -*** Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iteration are evenly (if possible) divided contiguously among the threads.

– ***dynamic*** - Loop iterations are divided into pieces of size chunk and then dynamically assigned to threads. When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

– ***guided*** - For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k(k>1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.

– ***runtime*** - The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause

– ***auto*** - The scheduling decision is made by the compiler and/or runtime system.

Static scheduling - 16 iterations, 4 threads:

Thread	0	1	2	3
*no chunk**	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

```
// Static schedule maps iterations to threads at
compile time
// static scheduling of matrix multiplication loops
#pragma omp parallel default (private) \
shared (a, b, c, dim) num_threads(4)
#pragma omp for schedule(static)
for(i=0; i < dim; i++)
{
 for(j=0; j < dim; j++)
 {
   c[i][j] = 0.0;
   for(k=0; j < dim; k++)
     c[i][j] += a[i][k]*b[k][j];
 }
}
```

Open MPI vs. OpenMP

Open MPI = Open Message Passing Interface. vs. **OpenMP** = Open Multi-Processing.

OpenMPI is an implementation of the Message Passing Interface (MPI) specification for writing distributed computing applications.

OpenMP is another API that adheres to the OpenMP specification for writing shared memory multi-process applications.

Both are used for writing high performance computing applications but with differences in approaches. MPI focuses on using message passing paradigms which is generally shared nothing. OpenMP focuses on shared memory paradigms.

https://www.quora.com/What-is-the-difference-between-OpenMP-and-Open-MPI | https://www.quora.com/What-are-the-differences-between-OpenMP-and-OpenMPI-What-is-better-and-easier-to-use-in-a-parallel-computing-project

Open MPI vs. OpenMP

Mark Hahn, computer guy at McMaster/Sharcnet/ComputeCanada, Answered Jun 19, 2019:

"**OpenMP** is a widely-used standard for shared-memory (threads) programming. The idea is that multiple cores/processors will operate inside the same process (address space), and so will need to perform locking, will need to divide work among them, etc.

OpenMPI is a widely used implementation of **MPI**, which is the standard for message-passing. Here, ***you have multiple separate processes, all running the same program, which communicate by sending and receiving messages with their peers. MPI provides a variety of message primitives: send/recv but also wait and various composite operations like reduction***.

If your project will **never** exceed one machine (host, server), then OpenMP may be what you need.

If you ***want to scale across multiple nodes*** (e.g. **Distributed System**), ***you definitely want*** **MPI** (which is also normally tuned to take advantage of a high-performance interconnect like Infiniband.)"

https://www.quora.com/What-is-the-difference-between-OpenMP-and-Open-MPI | https://www.quora.com/What-are-the-differences-between-OpenMP-and-OpenMPI-What-is-better-and-easier-to-use-in-a-parallel-computing-project

Open MPI vs. OpenMP

https://stackoverflow.com/questions/32464084/what-are-the-differences-between-mpi-and-openmp:

"**MPI** (with its implementations – e.g. **OpenMPI**) is a way ***to program on distributed memory devices***. This means that the parallelism occurs where every parallel process is working in its own memory space in isolation from the others.

You can think of it as: every bit of code you've written is executed independently by every process. The parallelism occurs because you tell each process exactly which part of the global problem they should be working on based entirely on their process ID.

OpenMP is a way ***to program on shared memory devices***. This means that the parallelism occurs where every parallel thread has access to all of your data.
You can think of it as: parallelism can happen during execution of a specific for loop by splitting up the loop among the different threads.

The way in which you write an OpenMP and MPI program, of course, is also very different."

https://www.quora.com/What-is-the-difference-between-OpenMP-and-Open-MPI | https://www.quora.com/What-are-the-differences-between-OpenMP-and-OpenMPI-What-is-better-and-easier-to-use-in-a-parallel-computing-project

Open MPI vs. OpenMP

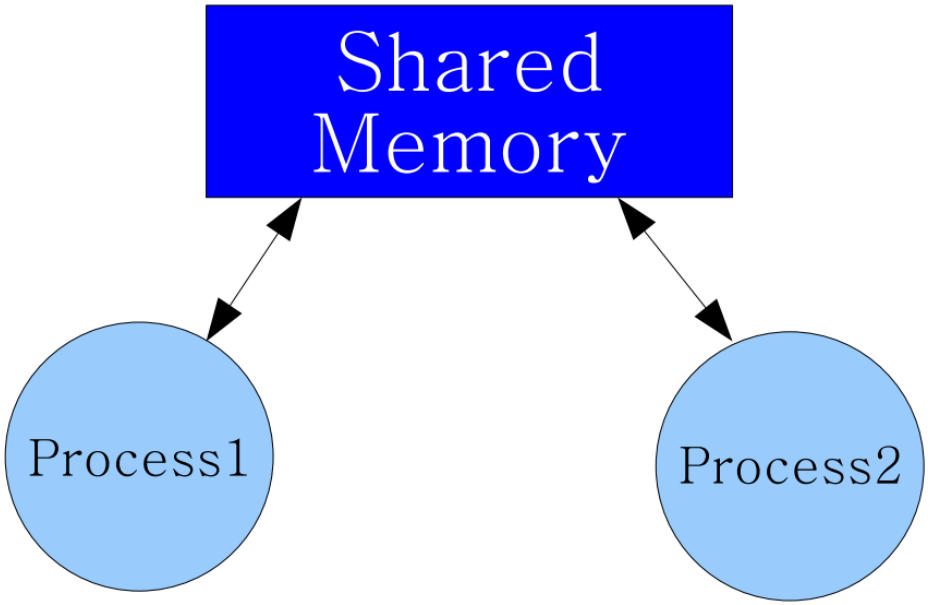
- Message Passing Interface (MPI)

  - MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.

Open MPI vs. OpenMP

- Open Multi Processing (OpenMP)

  - OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.



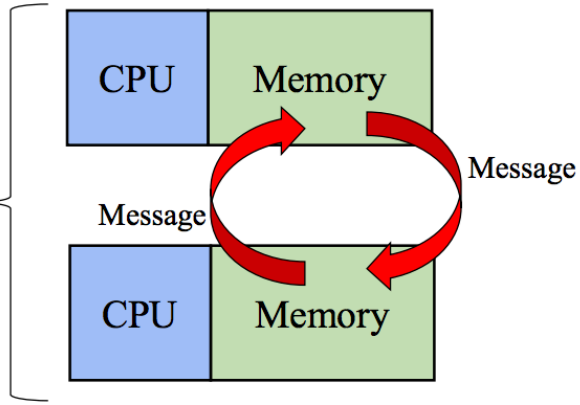
MPI vs. OpenMP

MPI	OpenMP
Distributed memory model	Shared memory model
on Distributed network	on Multi-core processors
Message based	Directive based
Flexible and expressive	Easier to program and debug

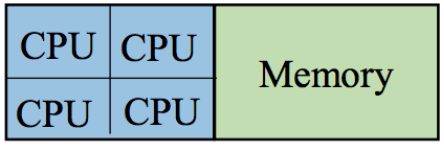
MPI and OpenMP

- MPI – Designed for distributed memory
  - Multiple systems
  - Send/receive messages
- OpenMP – Designed for shared memory
  - Single system with multiple cores
  - One thread/core sharing memory
- C, C++, and Fortran
- There are other options
  - Interpreted languages with multithreading
    - Python, R, matlab (have OpenMP & MPI underneath)
  - CUDA, OpenACC (GPUs)
  - Pthreads, Intel Cilk Plus (multithreading)
  - OpenCL, Chapel, Co-array Fortran, Unified Parallel C (UPC)

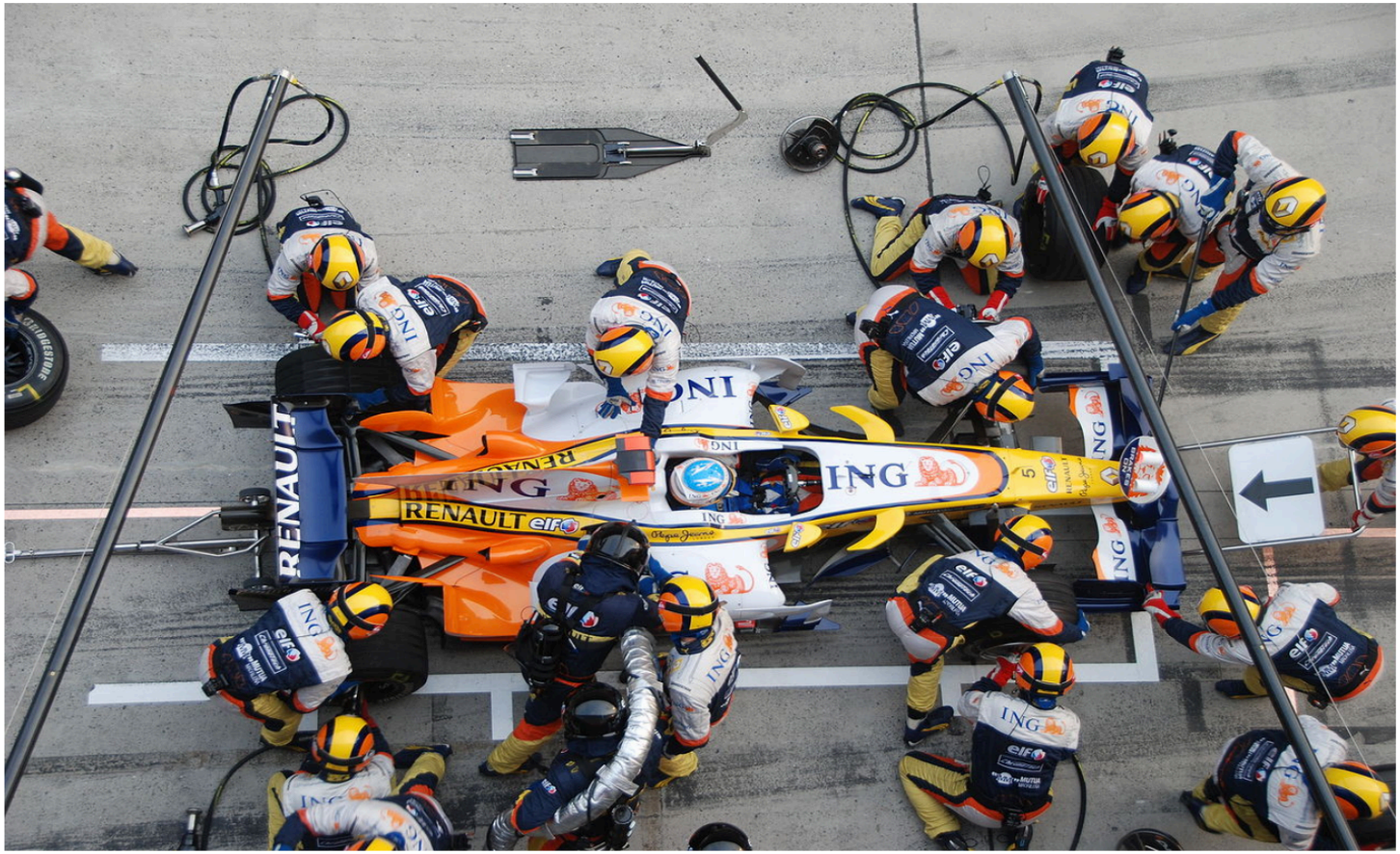
MPI



OpenMP



Parallel Programming Analogy



Source: Wikapedia.org

Open MPI Intro

- Parallel programming requires work:
    - Code modification – always
    - Algorithm modification – often
    - New sneaky bugs – you bet
- Speedup limited by many factors – e.g. CPU speed, BUS speed, Network bandwidth, RAM size availability, etc.

Ex. – Your program takes 20 days to run

- 95% can be parallelized
- 5% cannot (serial)
- What is the fastest this code can run?
    - As many CPU's as you want!

As you consider parallel programming understanding the underlying architecture is important

- Performance is affected by hardware configuration:
    - Memory or CPU architecture
    - Numbers of cores/processor
    - Network speed and architecture

Open MPI Intro

Message Passing Interface

- Standard
  - MPI-1 – Covered in intro books and tutorials
  - MPI-2 – Added features
  - MPI-3 – Even more cutting edge
  - **MPI-4 – Released in 2018 implementation of OpenMPI:**

https://www.open-mpi.org/software/ompi/v4.0/

* **MPI-5** Standard in development: https://github.com/open-mpi/ompi/wiki/5.0.x-FeatureList

- Distributed Memory
  - But can work on shared

- Multiple implementations exist
  - **Open MPI (work in our Linux VM and AWS EC2)**
  - MPICH
  - Many commercial (**Intel**, HP, etc..)

- ***Difference should only be in the compilation not development***
- C,C++, and Fortran

Open MPI Intro

An Interface Specification:

- **M P I = Message Passing Interface**
- MPI is a specification for the developers and users of message passing libraries.
- By itself, it is NOT a library - but rather the specification of what such a library should be.

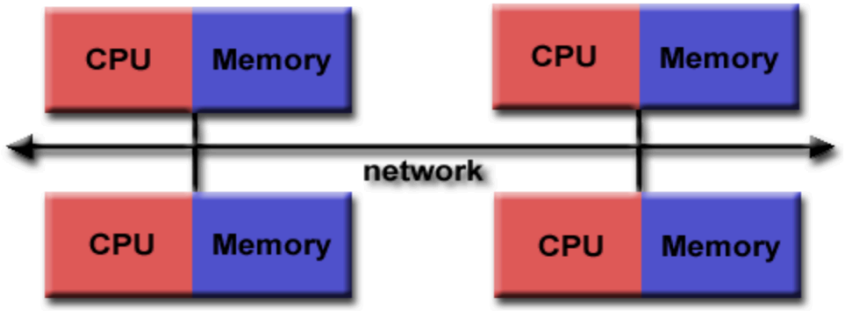
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  - Practical
  - Portable
  - Efficient
  - Flexible

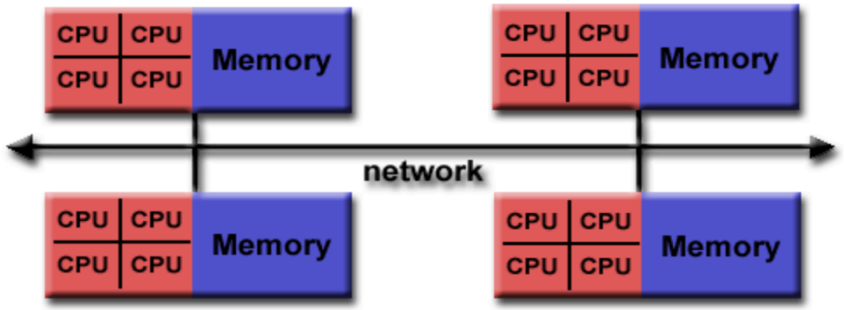
Open MPI Intro

Programming Model:

- Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.
- MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.



- Today, MPI runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid
- The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Open MPI Intro

Multiple Implementations:

- Although the MPI programming interface has been standardized, actual library implementations will differ.
- For example, just a few considerations of many:
  - Which version of the MPI standard is supported?
  - Are all of the features in a particular MPI version supported?
  - Have any new features been added?
  - What network interfaces are supported?
  - How are MPI applications compiled?
  - How are MPI jobs launched?
  - Runtime environment variable controls?
- MPI library implementations on LC systems vary, as do the compilers they are built for. These are summarized in the table below:

MPI Library	Where?	Compilers
MVAPICH	Linux clusters	GNU, Intel, PGI, Clang
Open MPI	Linux clusters	GNU, Intel, PGI, Clang
Intel MPI	Linux clusters	Intel, GNU
IBM Spectrum MPI	Coral Early Access and Sierra clusters	IBM, GNU, PGI, Clang

- Each MPI library is briefly discussed in the following sections, including links to additional detailed information.

Open MPI Intro

Cornell University
Center for Advanced Computing

Overview Introduction

- What is message passing?
  - Sending and receiving messages between *tasks* or *processes*
  - Includes performing operations on data in transit and synchronizing tasks
- Why send messages?
  - Clusters have distributed memory, i.e. each process has its own address space and no way to get at another's
- How do you send messages?
  - Programmer makes use of an Application Programming *Interface* (API)
    - In this case, MPI.
  - MPI specifies the functionality of high-level communication routines
  - MPI's functions give access to a low-level *implementation* that takes care of sockets, buffering, data copying, message routing, etc.

Open MPI Intro

Overview API for Distributed Memory Parallelism

- Assumption: processes do not see each other's memory
  - Some systems overcome this assumption
    - GAS (Global Address Space) abstraction and variants
- Communication speed is determined by some kind of network
  - Typical network = switch + cables + adapters + software stack…
- Key: the *implementation* of MPI (or any message passing API) can be optimized for any given network
  - Expert-level performance
  - No code changes required
  - Works in shared memory, too

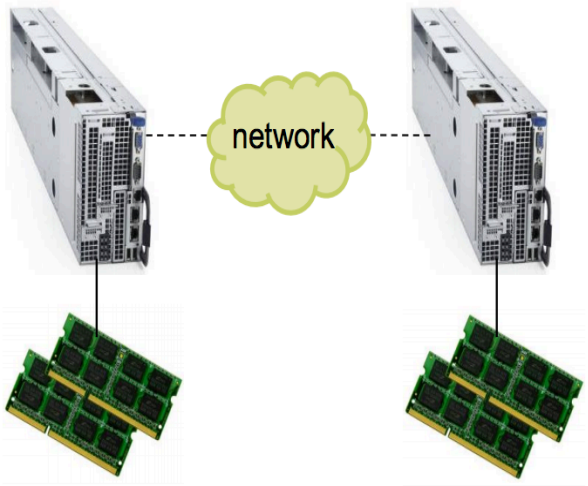


Image of Dell PowerEdge C8220X: http://www.theregister.co.uk/2012/09/19/dell_zeus_c8000_hyperscale_server/

Open MPI Intro

Overview Why Use MPI?

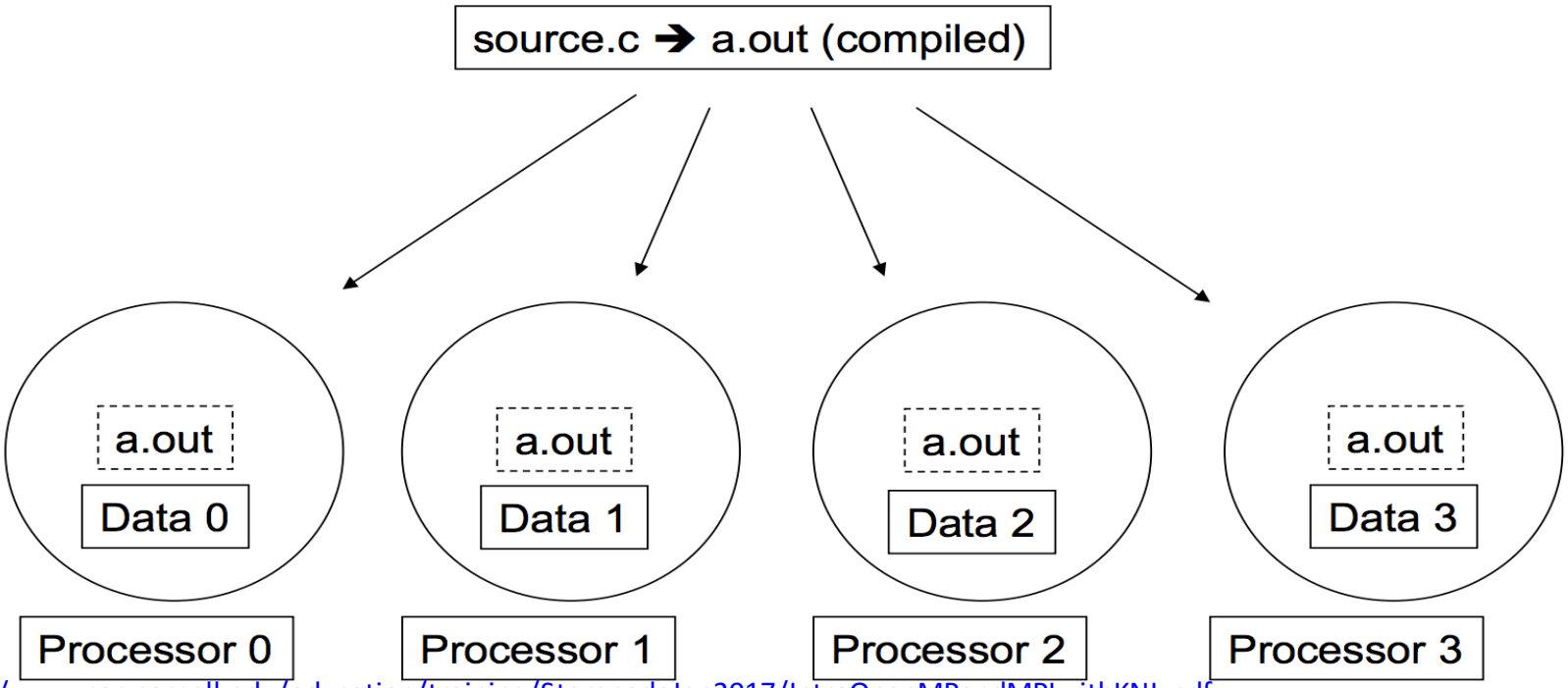
- MPI is a de facto standard for distributed memory computing
  - Public domain versions are easy to install
  - Vendor-optimized version are available on most hardware
- MPI is "tried and true"
  - MPI-1 was released in 1994, MPI-2 in 1996, and MPI-3 in 2012.
- MPI applications can be fairly portable
- MPI is a good way to learn parallel programming
- MPI is expressive: it can be used for many different models of computation, therefore can be used with many different applications
- MPI code is efficient (though some think of it as the "assembly language of parallel processing")
- MPI has freely available implementations (e.g., MPICH, OpenMPI)

Open MPI Intro

MPI and Single Program, Multiple Data (SPMD)

- One source code is written
- Same program runs multiple times, but each time with different data
- With MPI
  - Code can have conditional execution based on which processor is executing the copy: choose data
  - All copies of code are started simultaneously and may communicate and sync with each other periodically
  - Conclusion: MPI allows more SPMD programs than embarrassingly parallel applications

SPMD Programming Model



Open MPI Intro

Basics

Minimal Code Example: hello_mpi.c

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
main(int argc, char **argv)
{
   char message[20];
   int i, rank, size, tag = 99;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
      strcpy(message, "Hello, world!");
      for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
   } else {
      MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   }
   printf("Message from process %d : %.13s\n", rank, message);
   MPI_Finalize();
}
```

Open MPI Intro

Include MPI Header File

Start of Program

(Non-interacting Code)

Initialize MPI

Run Parallel Code &
Pass Messages

End MPI Environment

(Non-interacting Code)

End of Program

Open MPI Intro

Include MPI Header File

Start of Program

(Non-interacting Code)

Initialize MPI

Run Parallel Code & Pass Messages

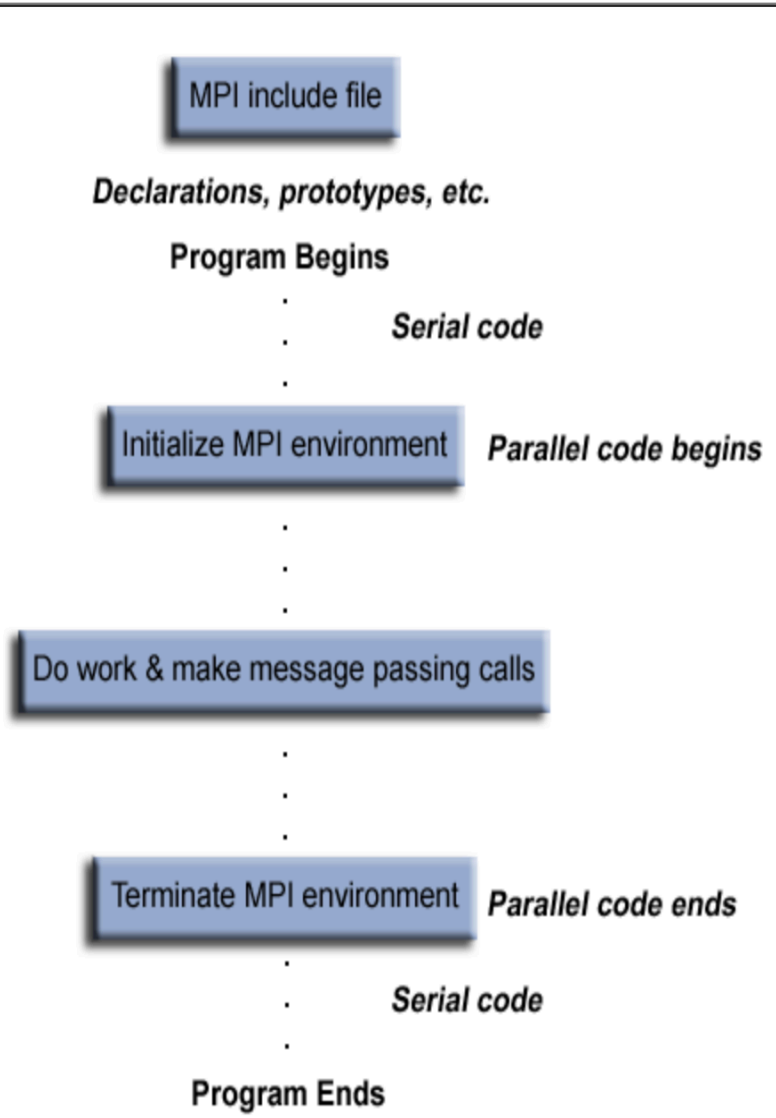
End MPI Environment

(Non-interacting Code)

End of Program

```
#include <mpi.h>

int main (int argc, char *argv[])
{

MPI_Init(&argc, &argv);

.
.     // Run parallel code
.

MPI_Finalize(); // End MPI Envir

return 0;
}
```

Open MPI Intro – General MPI Program Structure




```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

Basic Environment

```
MPI_Init(&argc, &argv)
```

- Initializes MPI environment
- Must be called in every MPI program
- Must be first MPI call
- Can be used to pass command line arguments to all

```
MPI_Finalize()
```

- Terminates MPI environment
- Last MPI function call

Communicators & Rank

- MPI uses objects called communicators
  - Defines which processes can talk
  - Communicators have a size
- MPI_COMM_WORLD
  - Predefined as ALL of the MPI Processes
  - $Size = N_{procs}$
- Rank
  - Integer process identifier
  - $0 \leq Rank < Size$

Basic Environment Cont.

```
MPI_Comm_rank(comm, &rank)
```

- Returns the rank of the calling MPI process
- Within the communicator, comm
  - MPI_COMM_WORLD is set during Init(…)
  - Other communicators can be created if needed

```
MPI_Comm_size(comm, &size)
```

- Returns the total number of processes
- Within the communicator, comm

```
int my_rank, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```c
int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);
  // Find out rank, size
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // We are assuming at least 2 processes for this task
  if (world_size < 2) {
    fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
    MPI_Abort(MPI_COMM_WORLD, 1);
  }

  int number;
  if (world_rank == 0) {
    // If we are rank 0, set the number to -1 and send it to process 1
    number = -1;
    MPI_Send(
      /* data         = */ &number,
      /* count        = */ 1,
      /* datatype     = */ MPI_INT,
      /* destination  = */ 1,
      /* tag          = */ 0,
      /* communicator = */ MPI_COMM_WORLD);
  } else if (world_rank == 1) {
    MPI_Recv(
      /* data         = */ &number,
      /* count        = */ 1,
      /* datatype     = */ MPI_INT,
      /* source       = */ 0,
      /* tag          = */ 0,
      /* communicator = */ MPI_COMM_WORLD,
      /* status       = */ MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
  }
  MPI_Finalize();
}
```

```c
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

```c
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

Open MPI Intro

Basics

Minimal Code Example: hello_mpi.c

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
main(int argc, char **argv)
{
   char message[20];
   int i, rank, size, tag = 99;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0) {
      strcpy(message, "Hello, world!");
      for (i = 1; i < size; i++)
         MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
   } else {
      MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   }
   printf("Message from process %d : %.13s\n", rank, message);
   MPI_Finalize();
}
```

Open MPI Intro

Basics

Initialize and Close Environment

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  } else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Message from process %d : %.13s\n", rank, message);
  MPI_Finalize();
}
```

Initialize MPI environment

An implementation may also use this call as a mechanism for making the usual argc and argv command-line arguments from “main” available to all tasks (C language only).

Close MPI environment

Open MPI Intro

Basics

Query Environment

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  } else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Message from process %d : %.13s\n", rank, message);
  MPI_Finalize();
}
```

Returns number of processes
This, like nearly all other MPI functions, must be called after MPI_Init and before MPI_Finalize. Input is the name of a communicator (MPI_COMM_WORLD is the global communicator) and output is the size of that communicator.

Returns this process' number, or rank
Input is again the name of a communicator and the output is the rank of this process in that communicator.

Open MPI Intro

Basics Pass Messages

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
main(int argc, char **argv)
{
  char message[20];
  int i, rank, size, tag = 99;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    strcpy(message, "Hello, world!");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  } else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
  printf("Message from process %d : %.13s\n", rank, message);
  MPI_Finalize();
}
```

Send a message
Blocking send of data in the buffer.

Receive a message
Blocking receive of data into the buffer.

Basics Compiling MPI Programs

- Generally, one uses a special compiler or wrapper script
  - Not defined by the standard
  - Consult your implementation
  - Correctly handles include path, library path, and libraries
- On Stampede 2, use MPICH-style wrappers (the most common)

```
mpicc -o foo foo.c
mpicxx -o foo foo.cc
mpif90 -o foo foo.f
```

(also mpif77)

  - Choose compiler+MPI with “module load” (default, Intel17+Intel MPI)

Open MPI Intro

Basics

Running MPI Programs

- To run a simple MPI program, use MPICH-style commands

  (Can't do this on login nodes!)

  `mpirun -n 4 ./foo` (usually mpirun is just a soft link to…)

  `mpiexec -n 4 ./foo`

- Some options for running

  `-n` -- states the number of MPI processes to launch

  `-wdir <dirname>` -- starts in the given working directory

  `--help` -- shows all options for *mpirun*

Open MPI Intro

Types of Point-to-Point Operations:

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- There are different types of send and receive routines used for different purposes. For example:
  - Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send
  - Combined send/receive
  - "Ready" send
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

Open MPI Intro – Point 2 Point Communications

Blocking

- Only returns after completed
    - Receive: data has arrived and ready to use
    - Send: safe to reuse sent buffer
- Be aware of deadlocks
- Tip: Use when possible

Non-Blocking

- Returns immediately
    - Unsafe to modify buffers until operation is known to be complete
- Allows computation and communication to overlap
- Tip: Use only when needed

Open MPI Intro – Point 2 Point Communications

```
MPI_Send(&buf, count, datatype, dest, tag, comm)
```

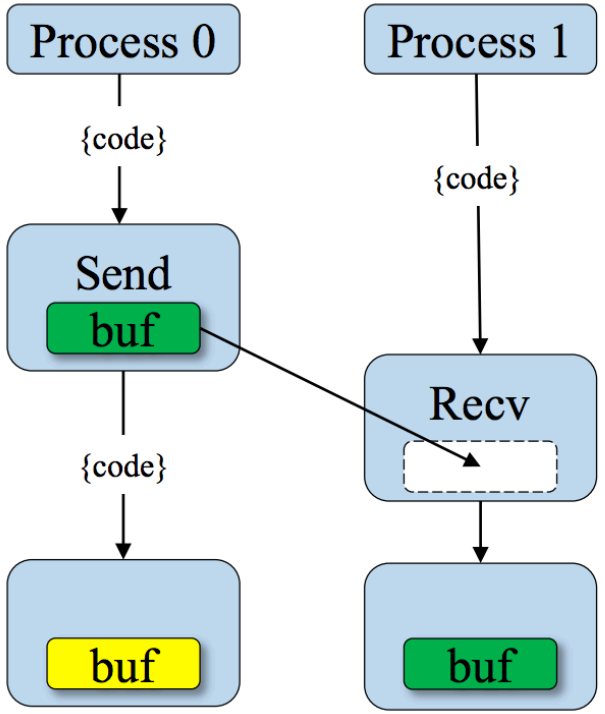
- Send a message
- Returns only after buffer is free for reuse (Blocking)

```
MPI_Recv(&buf, count, datatype, source, tag, comm, &status)
```

- Receive a message
- Returns only when the data is available
  - Blocking

```
MPI_SendRecv(...)
```

- Two way communication
- Blocking



Open MPI Intro

Blocking vs. Non-blocking:

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- **Blocking:**
  - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.
- **Non-blocking:**
  - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

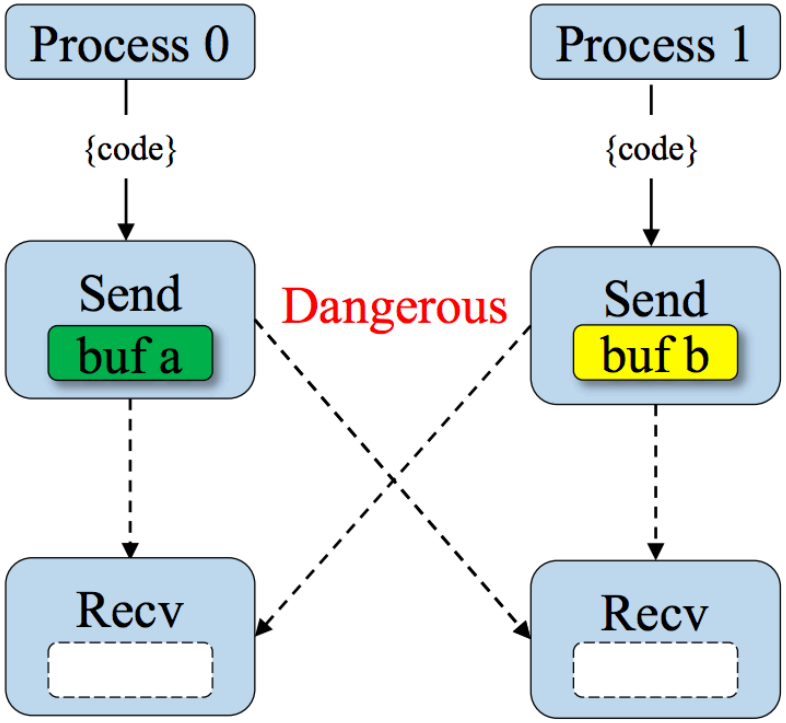
Blocking Send	Non-blocking Send

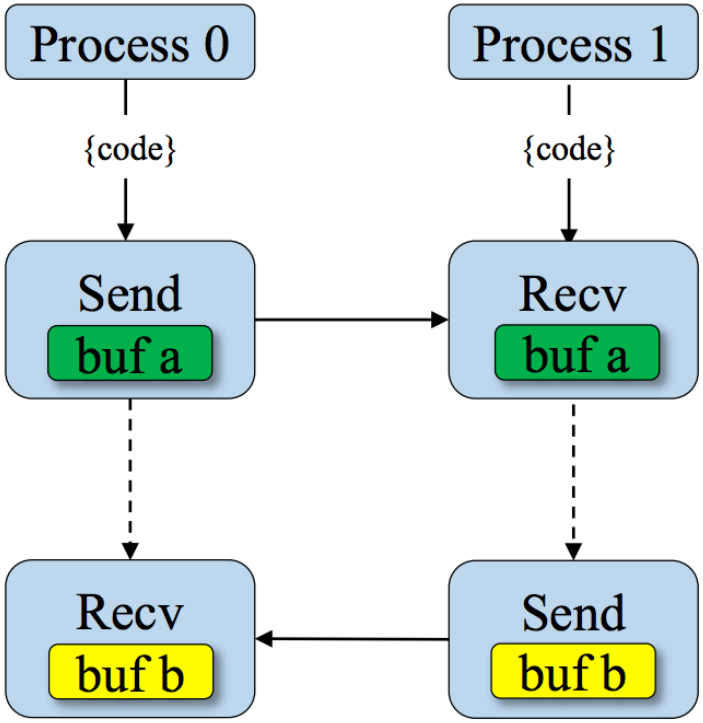
`myvar = 0;`
 	
`for (i=1; i<ntasks; i++) {`
`  task = i;`
`  MPI_Send (&myvar ... ... task ...);`
`  myvar = myvar + 2`
 	
`  /* do some work */`
 	
`  }`
	`myvar = 0;`
	
	`for (i=1; i<ntasks; i++) {`
	`  task = i;`
	`  MPI_Isend (&myvar ... ... task ...);`
	`  myvar = myvar + 2;`
	
	`  /* do some work */`
	
	`  MPI_Wait (...);`
	`  }`

Safe. Why?	**Unsafe. Why?**

Open MPI Intro – Point 2 Point Communications - DEADLOCK

- Blocking calls can result in deadlock
  - One process is waiting for a message that will never arrive
  - Only option is to abort the interrupt/kill the code (ctrl-c)
  - Might not always deadlock - depends on size of system buffer

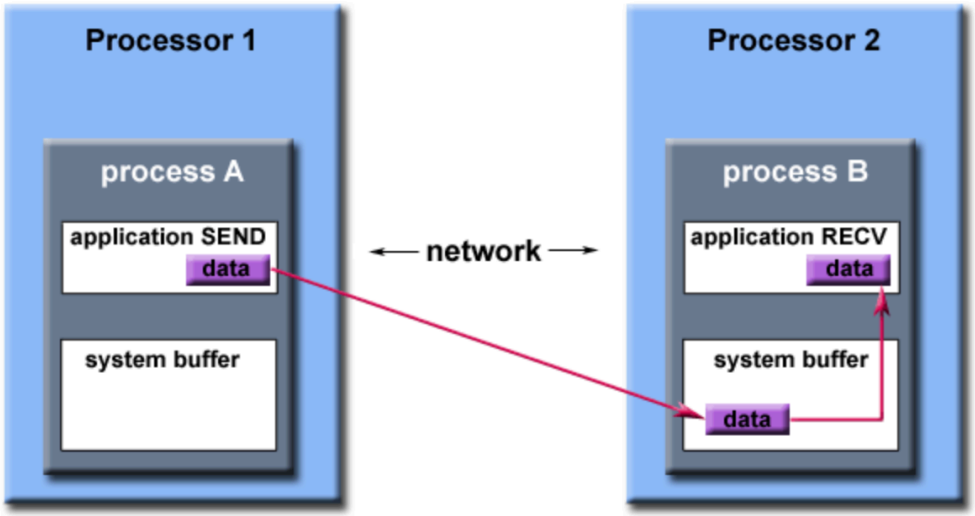




Open MPI Intro

Buffering:

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a **system buffer** area is reserved to hold data in transit. For example:



Path of a message buffered at the receiving process

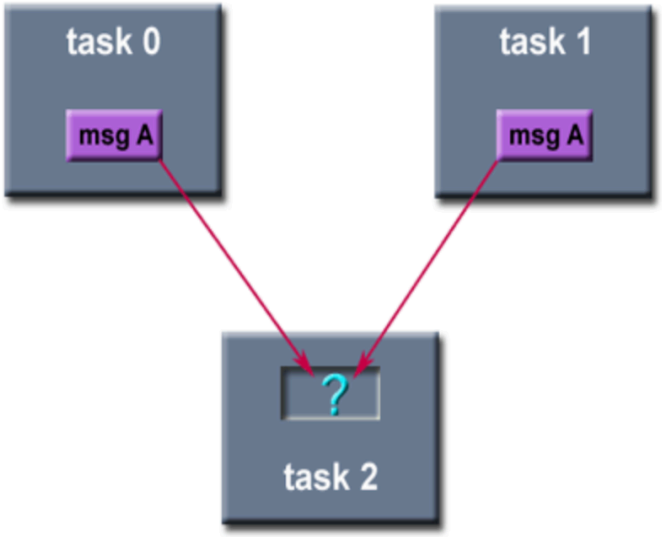
- System buffer space is:
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - Able to exist on the sending side, the receiving side, or both
  - Something that may improve program performance because it allows send - receive operations to be asynchronous.
- User managed address space (i.e. your program variables) is called the **application buffer**. MPI also provides for a user managed send buffer.

Open MPI Intro

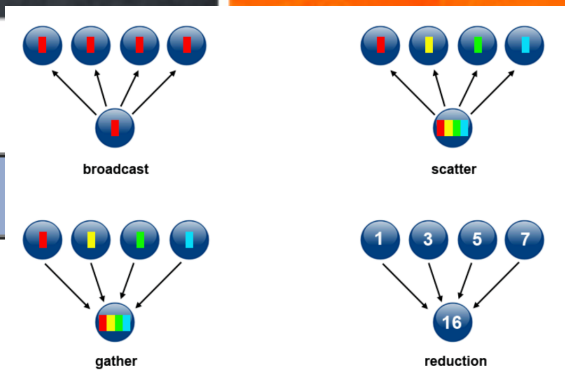
Order and Fairness:

- **Order:**
  - MPI guarantees that messages will not overtake each other.
  - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
  - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
  - Order rules do not apply if there are multiple threads participating in the communication operations.

- **Fairness:**
  - MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.



Open MPI Intro – Collective Communication



Collective Communication Routines

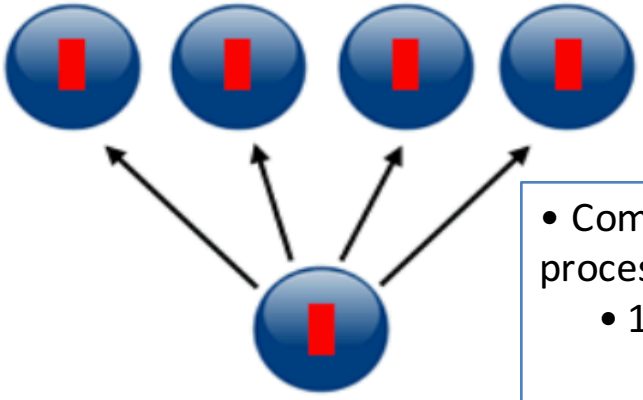
Types of Collective Operations:

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

Scope:

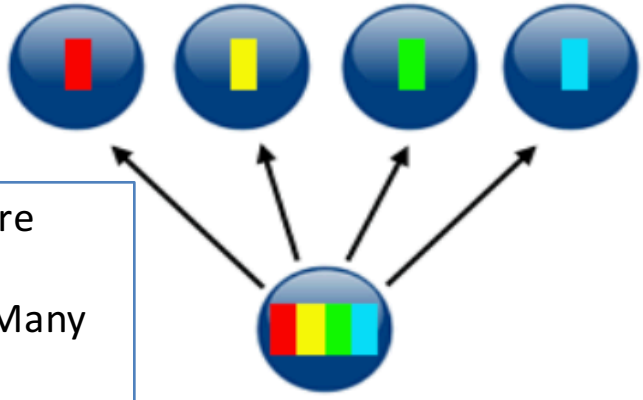
- Collective communication routines must involve **all** processes within the scope of a communicator.
  - All processes are by default, members in the communicator MPI_COMM_WORLD.
  - Additional communicators can be defined by the programmer. See the Group and Communicator Management Routines section for details.
- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Open MPI Intro – Collective Communication

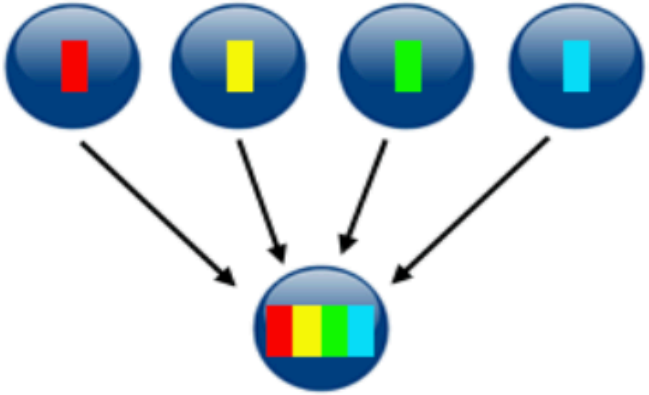


broadcast

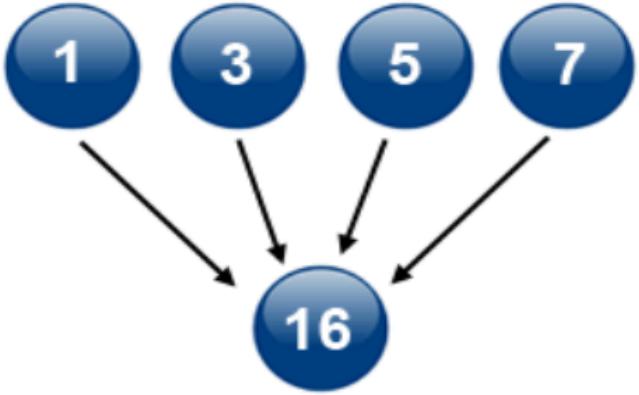
- Communication between 2 or more processes:
  - 1-to-Many, Many-to-1, Many-Many
- All processes call the same function with same arguments
- Data sizes must match
- Routines are blocking (MPI-1)



scatter



gather

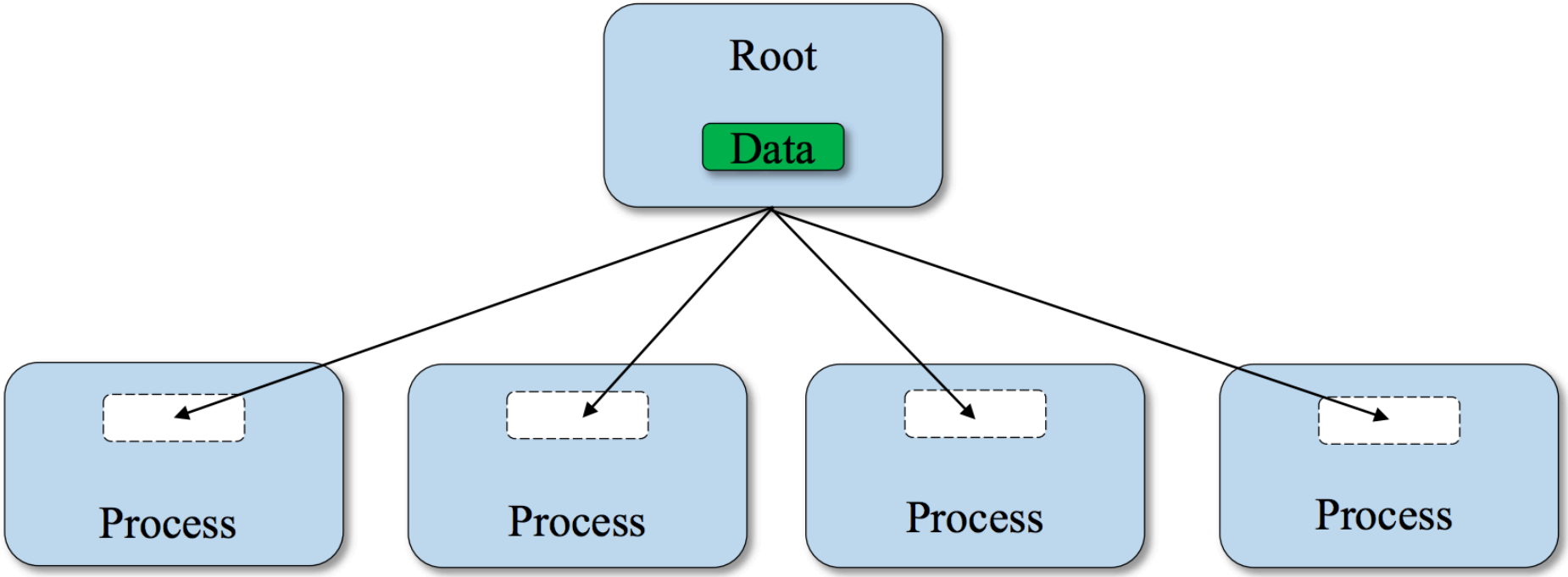


reduction

Open MPI Intro – Collective Communication

```
MPI_Bcast(&buffer, count, datatype, root, comm)
```

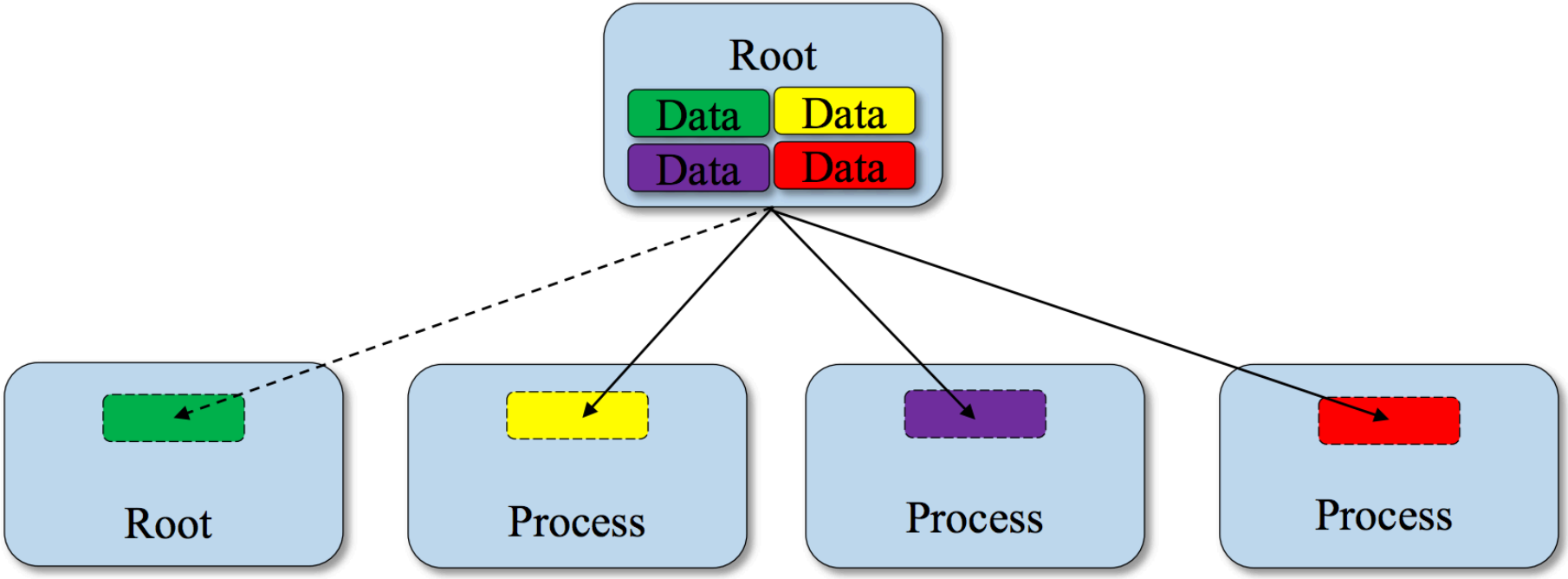
- Broadcasts a message from the root process to all other processes
- Useful when reading in input parameters from file



Open MPI Intro – Collective Communication

```
MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf,
            recvcnt, recvtype, root, comm)
```

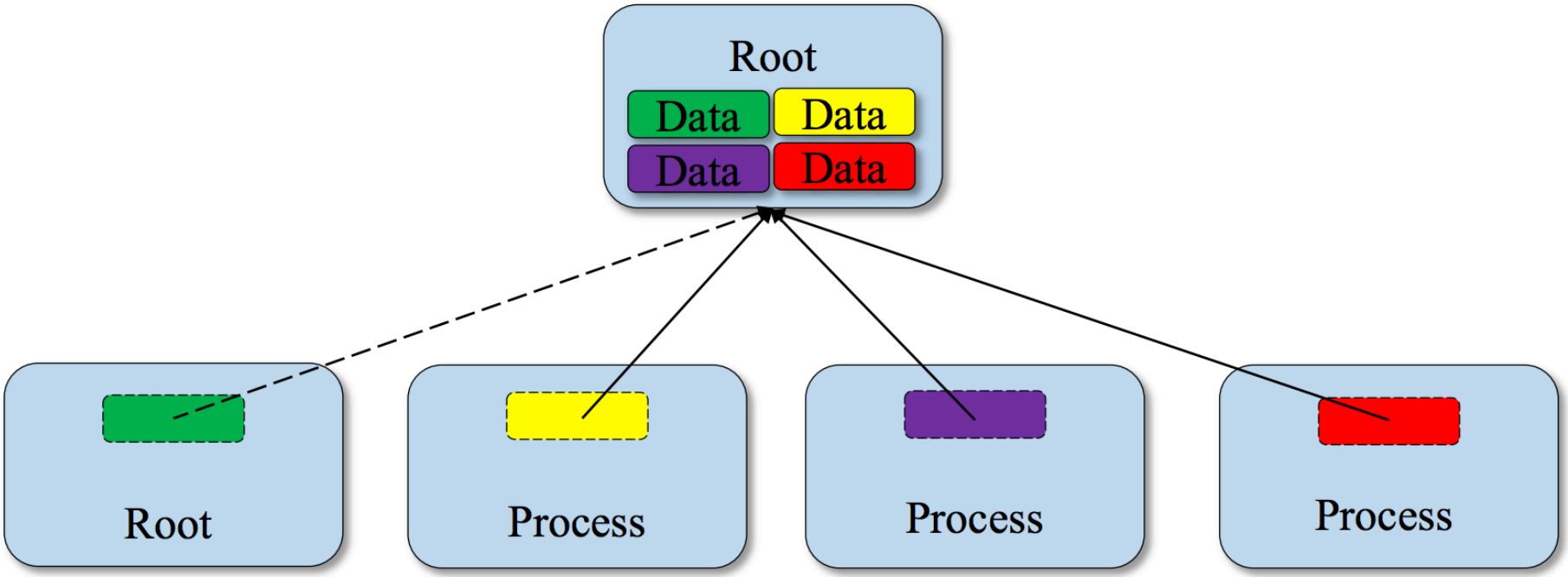
- Sends individual messages from the root process to all other processes



Open MPI Intro – Collective Communication

```
MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf,
           recvcnt, recvtype, root, comm)
```

- Opposite of Scatter

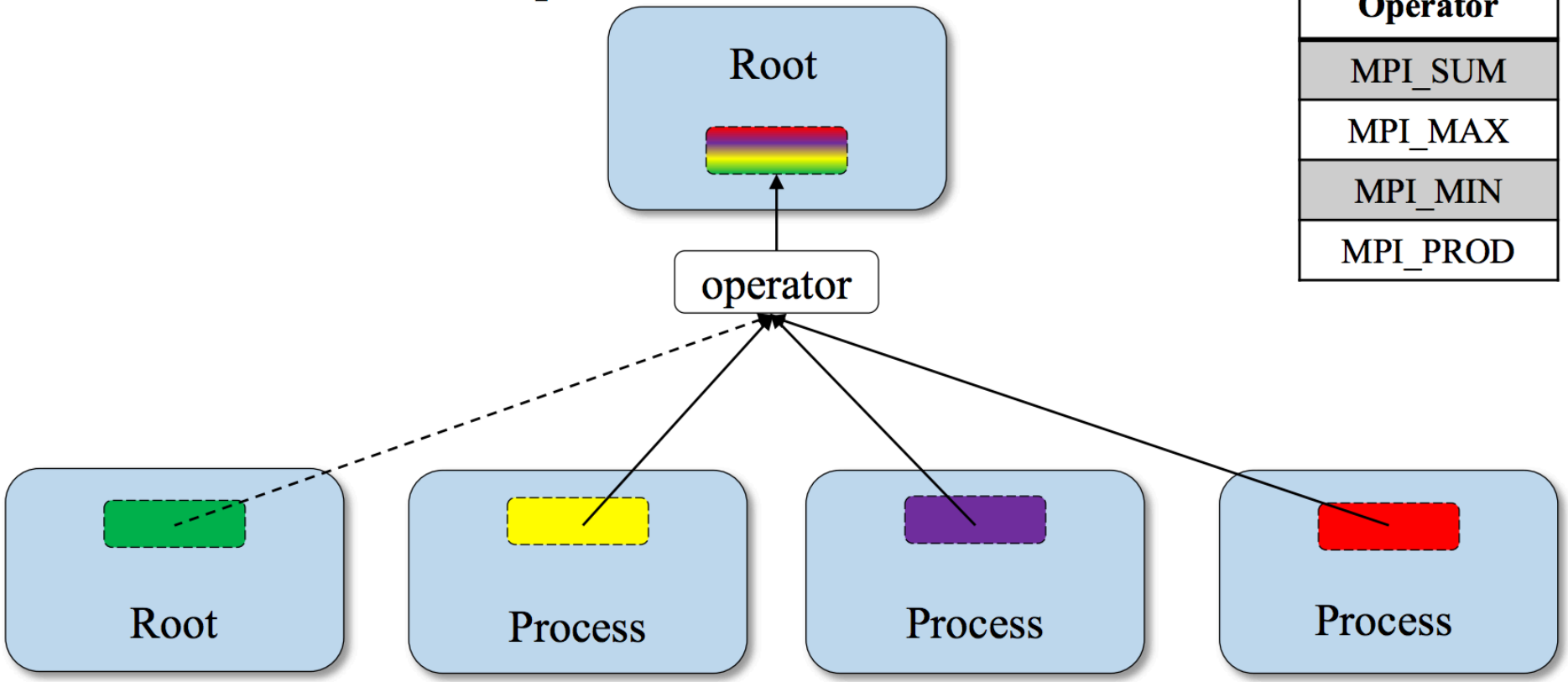


Open MPI Intro – Collective Communication

```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype,
           mpi_operation, root, comm)
```

- Applies reduction operation on data from all processes
- Puts result on root process

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

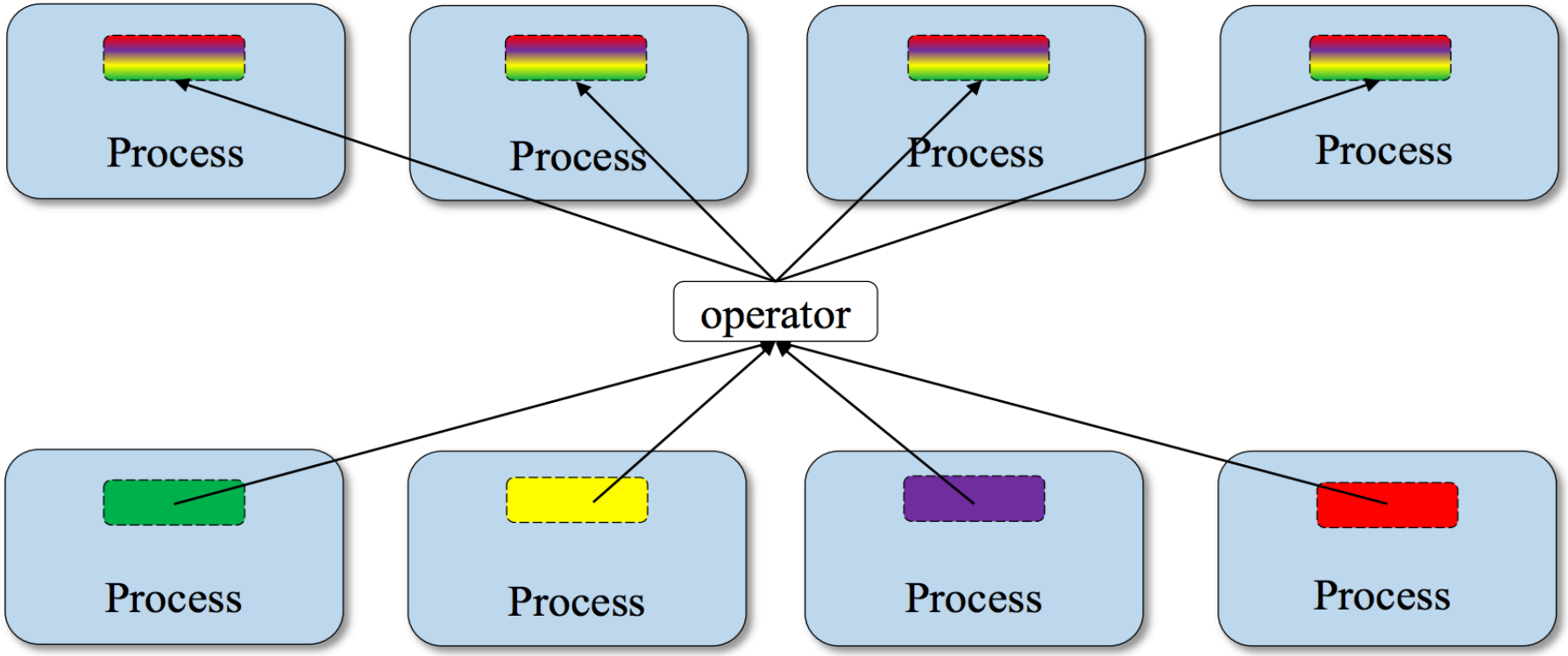


Open MPI Intro – Collective Communication

```
MPI_Allreduce(&sendbuf, &recvbuf, count,
              datatype, mpi_operation, comm)
```

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

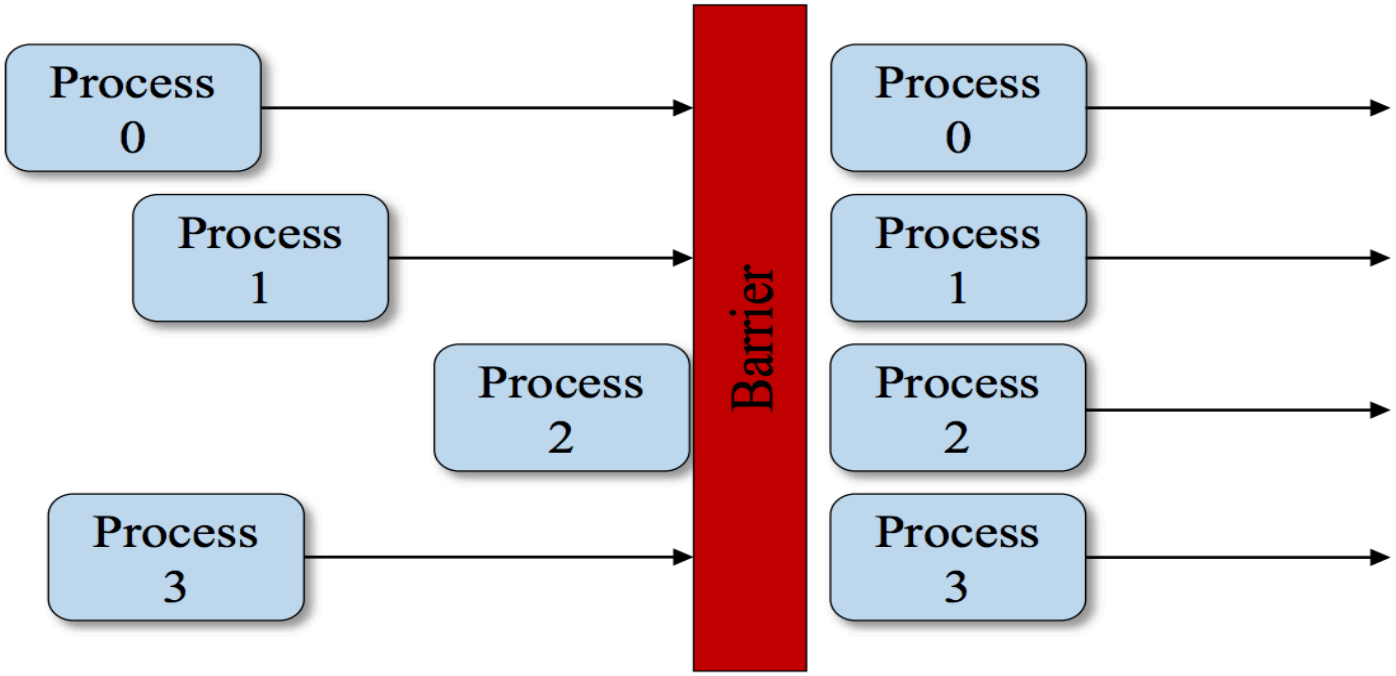
- Applies reduction operation on data from all processes
- Stores results on all processes



Open MPI Intro – Collective Communication

```
MPI_Barrier(comm)
```

- Process synchronization (blocking)
  - All processes forced to wait for each other
- Use only where necessary
  - Will reduce parallelism



Open MPI Intro – Collective Communication

Routine	Purpose/Function
MPI_Init	Initialize MPI
MPI_Finalize	Clean up MPI
MPI_Comm_size	Get size of MPI communicator
MPI_Comm_Rank	Get rank of MPI Communicator
MPI_Reduce	Min, Max, Sum, etc
MPI_Bcast	Send message to everyone
MPI_Allreduce	Reduce, but store result everywhere
MPI_Barrier	Synchronize all tasks by blocking
MPI_Send	Send a message (blocking)
MPI_Recv	Receive a message (blocking)
MPI_Isend	Send a message (non-blocking)
MPI_Irecv	Receive a message (non-blocking)
MPI_Wait	Blocks until message is completed

Open MPI Intro – Collective Communication

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Open MPI Intro – Collective Communication

C Language - Collective Communications Example

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #define SIZE 4
4
5  main(int argc, char *argv[])  {
6  int numtasks, rank, sendcount, recvcount, source;
7  float sendbuf[SIZE][SIZE] = {
8    {1.0, 2.0, 3.0, 4.0},
9    {5.0, 6.0, 7.0, 8.0},
10   {9.0, 10.0, 11.0, 12.0},
11   {13.0, 14.0, 15.0, 16.0}  };
12 float recvbuf[SIZE];
13
14 MPI_Init(&argc,&argv);
15 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
17
18 if (numtasks == SIZE) {
19   // define source task and elements to send/receive, then perform collective scatter
20   source = 1;
21   sendcount = SIZE;
22   recvcount = SIZE;
23   MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
24               MPI_FLOAT,source,MPI_COMM_WORLD);
25
26   printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
27          recvbuf[1],recvbuf[2],recvbuf[3]);
28   }
29 else
30   printf("Must specify %d processors. Terminating.\n",SIZE);
31
32 MPI_Finalize();
33 }
```

Sample program output:

```
rank= 0  Results: 1.000000 2.000000 3.000000 4.000000
rank= 1  Results: 5.000000 6.000000 7.000000 8.000000
rank= 2  Results: 9.000000 10.000000 11.000000 12.000000
rank= 3  Results: 13.000000 14.000000 15.000000 16.000000
```

Section Conclusions

OpenMP is NOT our goal. It is important for discipline “Parallel Programming”

OpenMPI is our target in terms of Parallel Computing over Distributed Systems

Java WS DEMO
for easy sharing

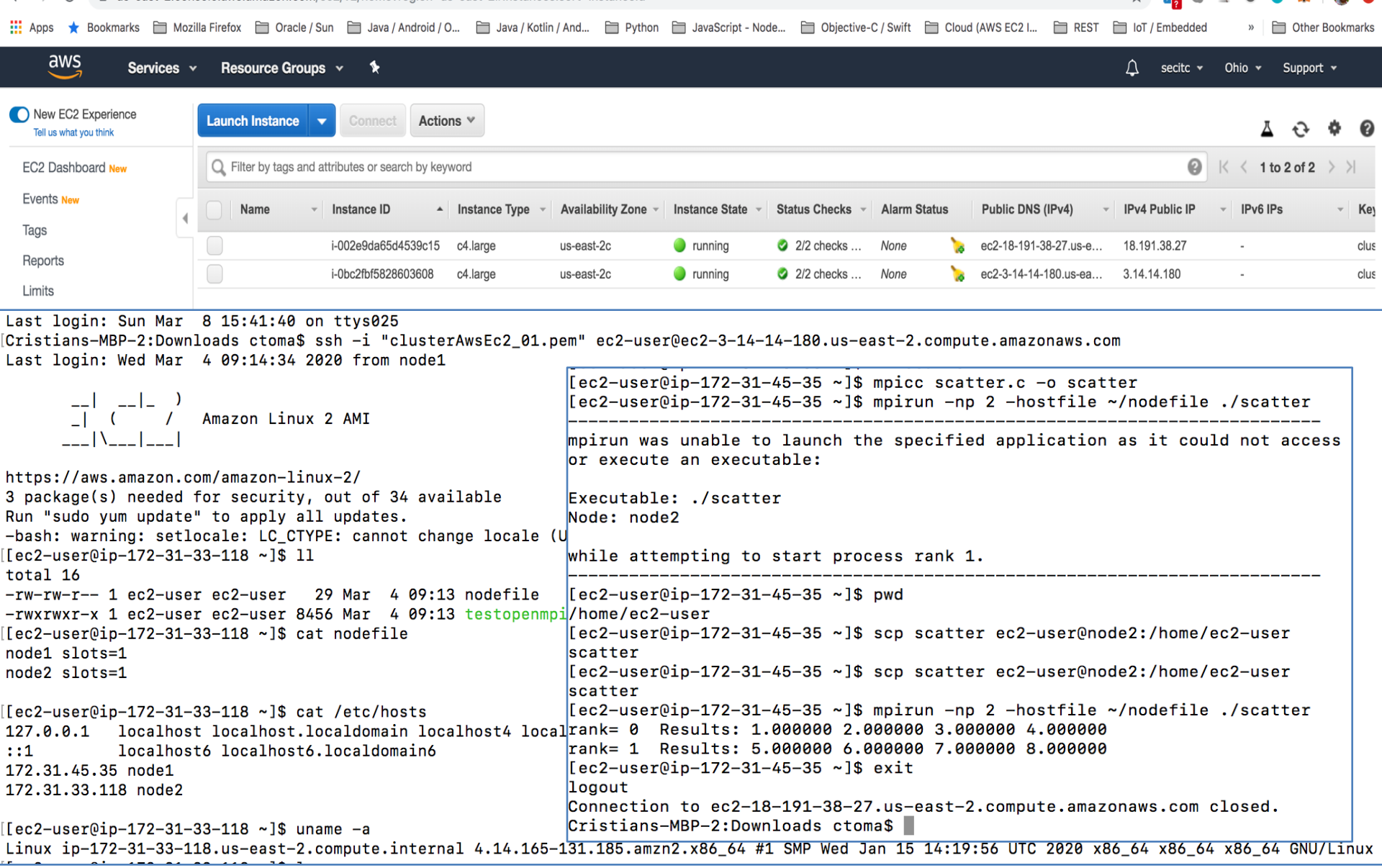
Distributed Application Development

Communicate & Exchange Ideas

Questions & Answers!

But wait…

There's More!



https://glennklockwood.blogspot.com/2013/04/quick-mpi-cluster-setup-on-amazon-ec2.html

Thanks!

DAD – Distributed Application Development
End of Lecture 8