Parallel Programming for Exascale

Jean-Marc GRATIEN[1]

[1] Department of Computer Science
IFP New Energy

November 4th 2020 / Master Data-HPCAI

Outline I

Outline II

Objectifs

Objectifs

- General Overview on Parallel Programming
- Introduce to tools for Exascale programming
- Introduction on Programming and Hardware Models
- Focus on various Parallel Programming for Shared Memory architecture:
    - Parallelization with pragma
    - Task Programming, DataFlow Programming
- Application : OpenMP, TBB, std::threads

Audience and Prerequisites

- Audience : computer science students
- Prerequisites :
  - sequential programming in C++
  - elementary algebraic math level (matrix vector operations)
  - image processing
- Material(Slide+TPs) available at :
  `https://drive.google.com/open?id=1HRx6qPRVYckY8H7KMdAcADWpI9iB-bl9`
  git clone `https://github.com/jgratien/ParallelProgrammingCourse.git`

Motivation

Exascale Challenge

Exascale Challenge

- What is the Exascale Challenge?
- Why?
- How?

Outline

Exascale Challenge

Exascale Challenge

- What is the Exascale Challenge?
- Why?
- How?

Exascale Challenge

What is the challenge

Exascale Challenge

- goal in 2008 :
  - top machines reach PetaPLOPS (10^{15} FLOPS)
  - by 2018, design computing systems capable of at least one exaFLOPS (10^{18} flops)
- current state 10 years later :
  - Japon : 2020, 415.5 petaflops, Fugaku powered by Fujitsu's 48-core A64FX (ARM)
  - China : 2018, 2 fastest computers in the world. First exascale computer, chiness one, will enter service by 2020 (school of computing at the National University of Defense Technology (NUDT)))
  - (USA) The Exascale Computing Project hopes to build an exascale computer by 2021 ;

Exascale Challenge?

Why such challenge

Why Exascale Challenge?

HPC resarch had in the past a real impact every body life

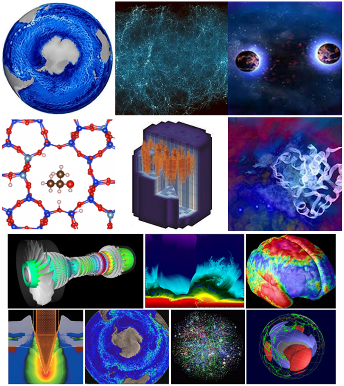
In 2008, the exascale challenge was plan to :

- improve national economic competitveness;
- advance scientific discovery;
- stengthen national security;

Why the Exascale Challenge?

Historically, HPC had an impact on many areas of science and engineering:

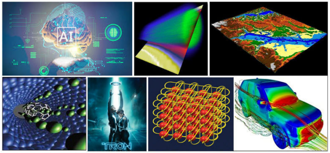
- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering - from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics
- Defense, Weapons
- Cosmology, Astropphysics
- ...



Why the Exascale Challenge?

Nowadays, Industrial and Commercial

- "Big Data", databases, data mining
- Artificial Intelligence (AI)
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial and economic modeling
- Management of national and multi-national corporations
- Advanced graphics and virtual reality, particularly in the entertainment
- industry Networked video and multi-media technologies
- Oil exploration
- Wind Energy



Roadmap fo Exascale

Various initiatives to achieve

- USA : Exascale Computing Project
- China : national plan for the next generation of high performance computers
- Europe : The CRESTA project (Collaborative Research into Exascale Systemware, Tools and Applications), the DEEP project (Dynamical ExaScale Entry Platform), and the project Mont-Blanc.[36] A major European project based on exascale transition is the MaX (Materials at the Exascale) project.
- Japon :

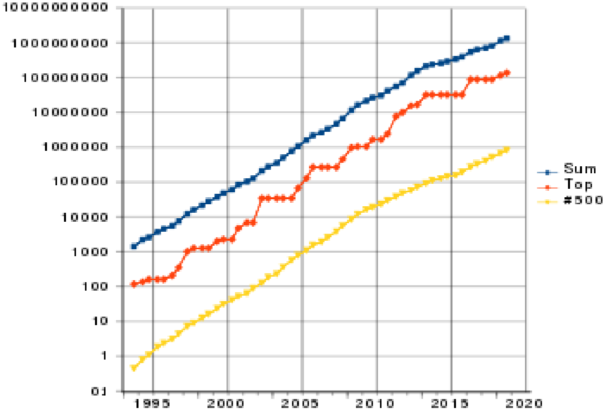
Exascale Challenge

Main issues to overcome

- Energy consumption reduction
  - impact on hardware design
  - heterogeneity (Computing Unit, Memory Units)
- trend on hardware design
  - impact on software design
  - lack of consensus
- Complexity management
  - software co-design
  - programming environment
  - abstractions, framework, layer architectures

Exascale Challenge
Trend

Top 500 evolution in 5 years

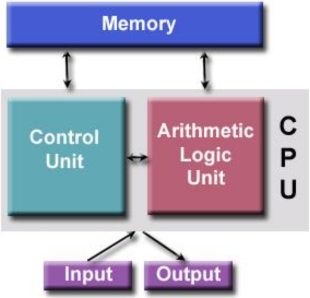


Outline

Hardware Architecture

Von Neumann Architecture

The Von Neumann Architecture :



For main components :

- Memory
- Control Unit
- Arithmetic Logic unit
- InputOutput

Hardware Architecture

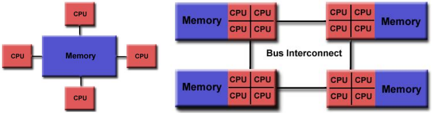
Flynn's Classical Taxonomy

- SISD : Single Instruction stream Single Data stream
- SIMD : Single Instruction stream Multiple Data stream
- MISD : Multiple Instruction stream Single Data stream
- MIMD : Multiple Instruction stream Multiple Data stream

Outline

Memory Architecture
Shared Memory

Ability for all processors to access all memory as global address space.



Classification :

- UMA : Uniform Memory Access
- NUMA : Non Uniform Access

Advantages :

- Global address space ;
- Data sharing between tasks ;

Disavantages :

- lack of scalability between memory and CPUs ;
- synchronisation management ;

Memory Architecture

Distributed Memory

Require a communication network to connect inter-processor memory

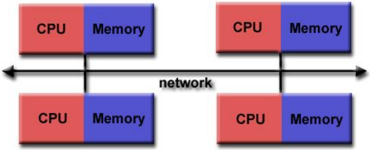
Local memory address

Advantages :

- Memory scalable with the number of processor ;
- Rapid access to local memory ;

Disavantages :

- requires communication ;
- lack of global address space ;
- non uniform memory access time.



Memory Architecture

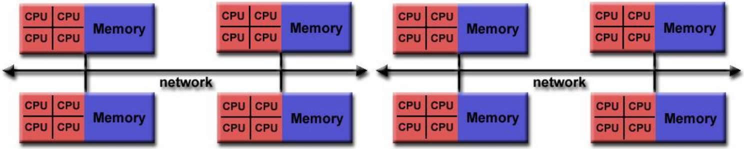
Hybrid Distributed-Shared Memory

Advantages :

- advantage of both systems ;

Disavantages :

- complexity management ;



Outline

Trend in hardware design

Heterogeneity

- Multi-scale process unit
  - VPU,Cores, processors, GP-GPU, acceleratos
  - various performance (enegy,speed,. . .)
  - need to manage load balancing
- Multi-scale memory unit
  - remote memory
  - multi-level local memory (cache L1,L2,L3,. . .), DRAM
  - Example : Latence (Core i7 Xeon 5500)

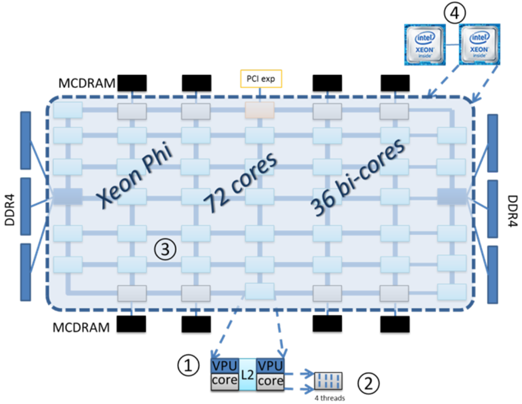
L1 CACHE hit	*simeq* 4 cycles	(2.1 - 1.2 ns)
L2 CACHE hit	$\simeq$ 10 cycles	(5.3 - 3.0 ns)
L3 CACHE hit unshared line	$\simeq$ 40 cycles	(21.4 - 12.0 ns)
L3 CACHE hit shared line in another core	$\simeq$ 65 cycles	(34.8 - 19.5 ns)
L3 CACHE hit modified by another core	$\simeq$ 65 cycles	(34.8 - 19.5 ns)
local DRAM	$\simeq$ 60 ns	
remote DRAM	$\simeq$ 100 ns	

  - need to manage coherency, synchronization, data movement

Trend in hardware design

Heterogeneity

Examples : Intel Knight Landing micro-architecture



Trend in hardware design

Heterogeneity

Examples : Intel Skylake Xeon micro-architecture

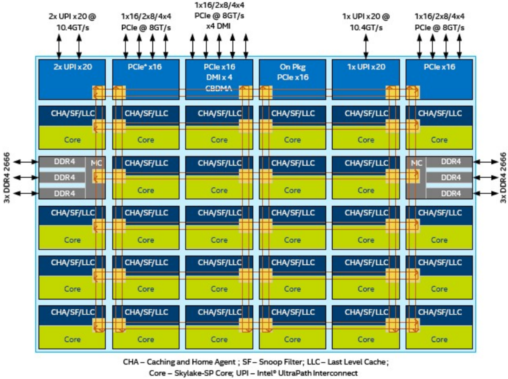
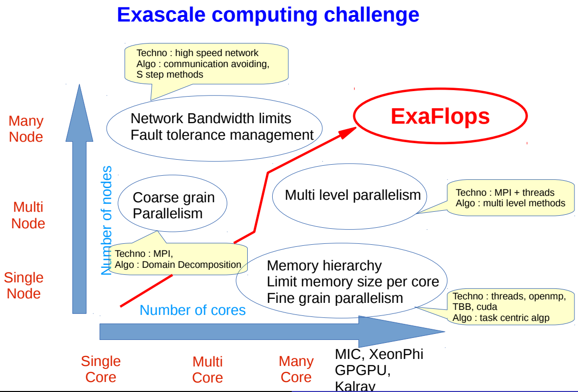


Figure: Intel Knight Landing micro-architecture

Trend in hardware design

Heterogeneity

Exascale RoadMap



Outline

Parallel Programming Models

Programming Models

Definition : an abstraction above hardware and memory architectures.

- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

Parallel Programming Models

Shared memory model

Parallel Programming Models

Threads model

Type of shared memory programming

A single "heavy weight" process can have multiple "light weight", concurrent execution paths

Implementation :

- POSIX Threads
  Specified by the IEEE POSIX 1003.1c standard (1995). C Language only. Part of Unix/Linux operating systems Library based Commonly referred to as Pthreads. Very explicit parallelism; requires significant programmer attention to detail.
- OpenMP
  Industry standard, jointly defined and endorsed by a group of major computer hardware and software vendors, organizations and individuals. Compiler directive based Portable / multi-platform, including Unix and Windows platforms Available in C/C++ and Fortran implementations Can be very easy and simple to use - provides for "incremental parallelism". Can begin with serial code.
- Microsoft threads
- Java, Python threads
- CUDA threads for GPUs

Parallel Programming Models

Message passing model

- MPI Message Passing Interface
- PVM Parallel Virtual Machine

Parallel Programming Models

Data Parallel Model

Partitioned Global Address Space (PGAS) model :

- Global adress space
- Data set are organized in vommon data structures

Current implementations:

- **Coarray Fortran**: a small set of extensions to Fortran 95 for SPMD parallel programming. Compiler dependent.
- **Unified Parallel C (UPC)**: an extension to the C programming language for SPMD parallel programming. Compiler dependent.
- **Global Arrays**: provides a shared memory style programming environment in the context of distributed array data structures. Public domain library with C and Fortran77 bindings.
- **X10**: a PGAS based parallel programming language being developed by IBM at the Thomas J. Watson Research Center.
- **Chapel**: an open source parallel programming language project being led by Cray.

Parallel Programming Models

Hybrid model

- MPI-X : MPI + OpenMP
- MPI + CUDA

Parallel Programming Models

Programming Models

- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

Outline

Parallel Algorithm Models

Various strategies

Strategy for dividing the data and processing method

- Data parallel model
- Task graph model
- Work pool model
- Master slave model
- Producer consumer or pipeline model
- Hybrid model

Outline

Parallel Random Access Machines

Here, n number of processors can perform independent operations on n number of data in a particular unit of time. This may result in simultaneous access of same memory location by different processors.
To solve this problem, the following constraints have been enforced on PRAM model

- Exclusive Read Exclusive Write (EREW) – Here no two processors are allowed to read from or write to the same memory location at the same time.
- Exclusive Read Concurrent Write (ERCW) – Here no two processors are allowed to read from the same memory location at the same time, but are allowed to write to the same memory location at the same time.
- Concurrent Read Exclusive Write (CREW) – Here all the processors are allowed to read from the same memory location at the same time, but are not allowed to write to the same memory location at the same time.
- Concurrent Read Concurrent Write (CRCW) – All the processors are allowed to read from or write to the same memory location at the same

Outline

Designing Parallel Programs

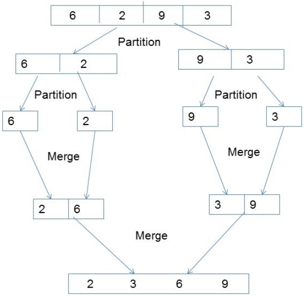
Parallel Design Pattern

Various Parallel Strategies:

- Divide and conquer
- Agglomeration
- Dynamic Programming
- Odd Even Communication
- Wavefront
- Reduction
- ...

Designing Parallel Programs
Parallel Design Pattern

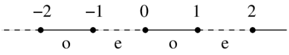
Divide and conquer : ParallelQuickSort



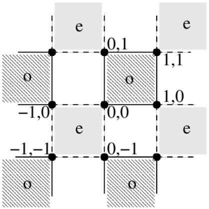
Designing Parallel Programs
Parallel Design Pattern

Odd Even Partition

(a)



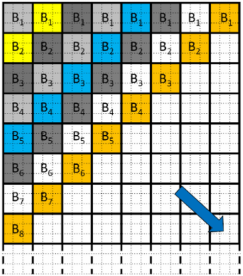
(b)



Designing Parallel Programs

Parallel Design Pattern

Wavefront



Mapping of Thread Blocks (B_i) to the Tiles within the Matrix

Outline

Designing Parallel Programs

Partitioning techniques

- Domain Decomposition
  - EDP
  - numerical methods based on meshes;
- Functional Decomposition
  - FFT, wave propagation
  - decomposition on direction, phases,. . .
- Monte Carlo methods
- . . .

Outline

Designing Parallel Programs

Communication and Synchronization

- Communication overhead
- Latency vs Bandwith
- Visibility
- Synchronous vs Asynchronous
- Scope of communications
- Complexity

Designing Parallel Programs

Data Dependencies

- Definition
- Data Flow
- Data movement
- ...

Designing Parallel Programs

Load Balancing

- Impact on parallel efficiency (barrier, synchronization,. . .)
- How to improve Load Balance :
  - Data distribution : dynamic partitioner
  - Work distribution : task scheduler

Designing Parallel Programs

Granularity

- Computation Communication Ratio
- Fine-grain Parallelism :
  - thread parallelism,SIMD, GP-GPU
  - easy for load balancing
  - low computation to communication ratio
- Coarse-grain Parallelism :
  - high computation to communication ratio rate
  - hard for load balancing

Designing Parallel Programs

Multi-level parallelism

New heterogeneous architectures imply to combine Coarse and Fine Grained parallelism

- Coarse grain parallelism :
  - cluster, socket level
  - reduce communication
- Fine-grain Parallelism :
  - thread parallelism,SIMD, GP-GPU
  - easy for load balancing

Outline

MPI Programming

History

- 1992 - 1994 : New group to define a standard API to implement Message Passing libraries
- MPI Forum : http://www.mpi-forum.org
- Purpose :
  - define a standard ;
  - implementation issues are not taken into account;
  - provide tools that ensure portability on distributed memory architecture

MPI Programming
History

- MPI 1 (1994) :
  - first C and Fortran interface
- Since : several normes
  - MPI 2 (1997),
    - new datatype constructor, langage interoperability
    - new functionalities, One side communication, MPI IO, dynamic process
    - Fortran, C++ bindings
  - MPI 3 (2012)
    - One side communication, non blocking collective communications
  - MPI 4 to come . . .
- Since : several implementations
  - MPICH, OpenMPI, MVAPICH, IntelMPI, . . .

MPI Programming

MPI

- Message Passing Interface
- it is a library (not a language) with a standrad API
- design to develop for distributed memory architecture
- based on a SPMD (Single Program Multiple Data) model
- a MPMD model is now available since MPI-2

MPI Programming

MPI

Outline

MPI Programming

MPI Concepts

HelloWord

```
#include <iostream>
#include <mpi.h>
int main(int argc, char* argv[])
{
   int nbTask ;
   int myRank ;
   MPI_Init(&argc, &argv) ;
   MPI_Comm_size(MPI_COMM_WORLD, &nbTask) ;
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank) ;
   std::cout«"HelloWord : rank="«myRank«" on nb tasks:"«nbTask«std::endl ;
   MPI_Finalize() ;
}
```

MPI Programming

MPI Concepts

- Compilation :

Compilation

```
mpicxx -o helloword.exe helloword.cc
```

- Execution :

Execution

```
mpirun -np <nb tasks> ./helloword.exe
```

MPI Programming

MPI Concepts : Basic primitives

- Header :

Execution

```
#include <mpi.h>
```

- Initialisation :

Compilation

```
int MPI_Init(int* argc, char*** argv);
```

- Finalization :

Execution

```
int MPI_Finalize();
```

MPI Programming

MPI Concepts : Communicator

- MPI Communicator

MPI Communicator

MPI_Comm comm ;

- define a static group of MPI process
- all the processes are in the predifined group : MPI_COMM_WORLD
- to get number of MPI proces in a MPI group:

MPI Group size

int MPI_Comm_size(MPI_ comm, int* size) ;

- to get an a process id in a MPI group

MPI process rang

int MPI_Commrank(MPI_ comm, int* rank) ;

MPI Programming

MPI Concepts : Data Types

- Basic types

MPI	C
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI Programming

MPI Concepts : Derived Data Types

- Derived types constructed from existing types

Define new Data Type

MPI_Type_contiguous(count,oldtype,newtype) ;
MPI_Type_vector(count,blocklenght,stride,oldtype,newtype) ;
MPI_Type_struct ;

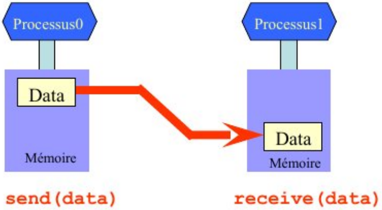
- Commit new datatype

Commit new Data Type

MPI_Type_commit(MPI_datatype *datatype) ;
MPI_Type_free(MPI_datatype *datatype) ;

MPI Programming

MPI Concepts : Communication



Communications features:

- Point to Point vs collective ;
- synchrone vs asynchrone ;
- various modes :
  - standard,
  - buffered,
  - synchronous,
  - ready.

MPI Programming

MPI Concepts : One to One

Standard

```
MPI_send(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm com) ;
MPI_recv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Status *status) ;
```

- data : buf, count, datatype
- source, dest : rank of send, recv MPI process (joker MPI_ANY_SOURCE)
- tag : message id (joker MPI_ANY_TAG)
- comm : MPI communicator
- status : MPI_Status object with message complementary info
- request : MPI_Request object to manage asynchrone communication

MPI Programming

MPI Concepts : One to One

- Asynchrone :

Asynchrone

MPI_Isend(. . . ,int dest, int tag, MPI_Comm comm, MPI_Request* request) ;
MPI_Irecv(. . . ,int source, int tag, MPI_Comm comm, MPI_Request* request);

- Synchrone :

Synchrone

MPI_Ssend(. . . ,int dest, int tag, MPI_Comm comm) ;
MPI_Srecv(. . . ,int source, int tag, MPI_Comm comm) ;

MPI Programming

MPI Concepts : One to One

- Ready :

Ready

MPI_Rsend(. . . ,int dest, int tag, MPI_Comm comm) ;
MPI_Rrecv(. . . ,int source, int tag, MPI_Comm comm) ;

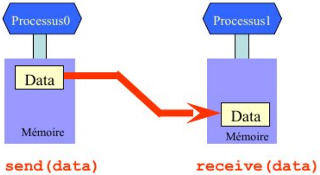
- Buffered :

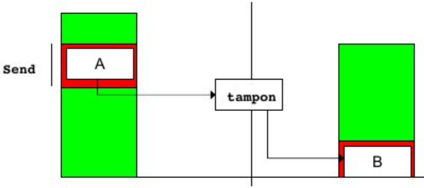
Buffered

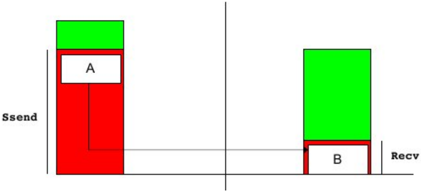
MPI_Bsend(. . . ,int dest, int tag, MPI_Comm comm) ;
MPI_Brecv(. . . ,int source, int tag, MPI_Comm comm) ;

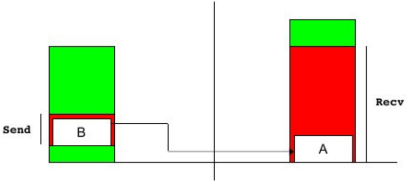
MPI Programming

MPI Concepts : One to One









MPI Programming

MPI Concepts : Collective

- Broadcast, Gather, Scatter, Alltoall ;

Broadcast

```
MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
    int root, MPI_Comm com) ;
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendatatype,
        void* recvbuf, int recvcount, MPI_Datatype sendatatype,
        int rout, MPI_Comm com) ;
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendatatype,
        void* recvbuf, int recvcount, MPI_Datatype sendatatype,
        int root,MPI_Comm com) ;
MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendatatype,
        void* recvbuf, int recvcount, MPI_Datatype sendatatype,
        MPI_Comm com) ;
```

MPI Programming

MPI Concepts : Collective

- Reduction, (MPI_Op : MPI_MAX, MPI_MIN, MPI_SUM,. . .);

Reduce

MPI_reduce(void *sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
    int root, MPI_OP op, MPI_Comm com) ;
MPI_allreduce(void *sendbuf, void* recvbuff, int count, MPI_Datatype
datatype,
    MPI_Op op, MPI_Comm com) ;

- Barrier ;

Barrier

int MPI_Barrier(MPI_Comm com) ;

MPI Programming

Specific MPI Issues

- Deadlock management
  - standard send recv have behaviour implementation dependent
  - need to check communication scheme
  - otherwise use asynchronous mode
- Communication Overlap
  - it is possible to overlap communication with computation
  - require asynchronous mode ;
  - postpone send or receive communication
  - require to manage communication buffer

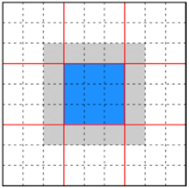
Outline

MPI Programming

MPI : Data partition, Domain Partition

- SPMD model implies Data Partition

- Partitioner :
  - Mesh, Graph, HyperGraph
  - Minimize communication
  - Ghost Data
    - Duplicate Data
    - Computation vs Communication
    - Synchronization to ensure coherency



Outline

Task Programming

Introduction to Task Programming

- Generic concepts, C++ since C++11
- OpenMP
- TBB
- Runtime System Tools
- ...

Outline

Task programming since C++11

Generic concepts

- Thread based Shared Memory Programming Model
- Synchronization
- Atomic operations

Task programming since C++11

Generic concepts

Thread based Shared Memory Programming Model :

Thread in C++

```
#include <iostream> // std::cout
#include <thread> // std::thread
void foo() {
  // do stuff..
. }
void bar(int x) {
  // do stuff...
}
int main() {
  std::thread first (foo); // spawn new thread that calls foo()
  std::thread second (bar,0); // spawn new thread that calls bar(0)
  // synchronize threads:
  first.join(); // pauses until first finishes
  second.join(); // pauses until second finishes
  std::cout « "foo and bar completed.";
  return 0;
}
```

Task programming since C++11

Generic concepts

Synchronization : Critical section

- Mutex :
  - *lock()*, *try_lock()*
  - *unlock()*;
- Lock concepts :
  - lock a mutex on construction
  - release mutex on destruction

Task programming since C++11

Generic concepts

Critical section with mutex

```
#include <thread> // std::thread
#include <mutex> // std::mutex
std::mutex mtx; // mutex for critical section
void doStuff (int n) {
  mtx.lock();
  ...
  mtx.unlock();
}
int main () {
  std::thread th1 (doStuff,50);
  std::thread th2 (doStuff,100);
  th1.join();
  th2.join();
  return 0;
}
```

Critical section with lock

```
#include <thread> // std::thread
#include <mutex> // std::mutex
std::mutex mtx; // mutex for critical section
void doStuff (int n) {
  std::lock_guard<std::mutex> lock(mtx) ;
  ...
  // automatic call destructor lock
}
int main () {
  std::thread th1 (doStuff,50);
  std::thread th2 (doStuff,100);
  th1.join();
  th2.join();
  return 0;
}
```

Task programming since C++11

Generic concepts

Atomic operations :

- *template< class T > struct atomic*;
- *operator++*, *operator–*;
- *operator+=*, *operator-=*;
- *store()*, *load()*;
- *exchange()*;
- *compare_exchange_weak()*;
- *compare_exchange_strong()*;

Outline

OpenMP

Introduction

OpenMP, Open Multi-Processing : standard for parallel programming on Shared Memory Architecture

- Thread based Shared Memory Programming Model
- directive based programming language
- portable
- C, C++, Fortran

OpenMP
History

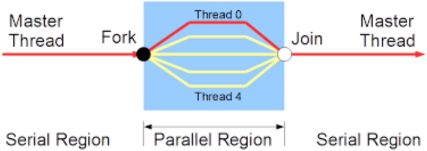
- 1991 : Parallel Computing Forum defines a set of diectived to parallelize Fortran Loops
- 1997 : OpenMP 1.0 standard for Fortran
- 1998 : standrad for CC++
- 2000 : OpenMP 2.0 standard for Fortran 1995
- 2008 : OpenMP 3.0 task concept
- 2013 : OpenMP 4.0 SIMD, accelerator
- 2017 : OpenMP 4.5 data mapping, doaccross,...

OpenMP
Principle

A standard API based on :

- directives (pragma interpreted at compile time)
- a library (dynamic functions executed at runtime)
- Environnement vatiables

Programming Model : Fork-Join Model



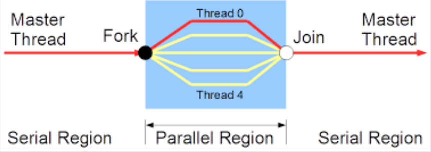
Memory Model : Shared Memory Model

- threads shared the main memory;
- each threads may manage a private memory

OpenMP
Principle

Programming Model

based on a Fork-Join Model :



Memory Model

based on a Shared Memory Model with threads

- threads shared the main memory;
- each threads may manage a private memory

OpenMP
Directives

DIRECTIVES CC++

#pragma omp *name* [*clause* [*clause*] . . .]
{
. . .
}

- *name* : directive name
- *clause* . . . : a liste of clauses
- the directive is applied to the following block

OpenMP
PARALLEL REGIONS

PARALLEL REGIONS

#pragma omp parallel [*clause* [*clause*]. . .]
{
// PARALLEL REGION
}

- define a parallel region
- the current thread creates a team of threads
- the current thread becomes the master of the team
- the size of the team depends on (by prority order):
  1. the clauses if,
  2. the clause num_threads,
  3. the function omp_set_num_threads()
  4. the environment variable OMP_NUM_THREADS
- the following block is executed by the threads of the team

OpenMP

PARALLEL REGIONS

FOR

#pragma omp for [*clause* [*clause*]. . .]
for(...)
...

SECTION

#pragma omp section [*clause* [*clause*]. . .]
...

SINGLE

#pragma omp single [*clause* [*clause*]. . .]
...

- implicite barrier at the end of parallel section
- unless clause *nowait*

OpenMP
PARALLEL REGIONS

clause *schedule*

schedule (*type*[,*chunk*])

iteration distribution policy

- **static** : iterations divided in blocks of size *chunk* and assigned to threads in a round-robbin mode
- **dynamic** : thread ask dynamically block of size *chunk*
- **runtime** : defined at runtime with the environment variable OMP_SCHEDULE
- **auto** : policy defined at compile or runtime time

OpenMP
Data Scope Attribute Clauses

data-clause(list)

list : variable list

- **private** : list of variable in private memory (original variables are duplicated)
- **firstprivate** : like private automatic initialisation from original variable
- **lastprivate** : like provite automatic update og original variable
- **shared** : list of shared variables (not duplicated)
- **default(shared|none)** : default scope of all variables
- **reduction(operator:list)** :
- **copyin(list)** : copy master variable value of list to other threads private copy
- **copyprivate(list)** : brodcast variable value of list from single section to other threads copy

OpenMP

Clause summary

Clause	Directives					
	parallel	for	sections	single	parallel for	parallel sections
if	x				x	x
private	x	x	x	x	x	x
shared	x	x			x	x
default	x				x	x
firstprivate	x	x	x	x	x	x
lastprivate	x	x	x	x	x	x
reduction	x	x	x		x	x
copyin	x				x	x
copyprivate				x		
schedule		x			x	
ordered		x		x		
nowait		x	x	x		

OpenMP

Synchronisation

Synchronization management :

- **barrier** : wait for all other team threads
- **ordered** : ensure that the following block respect sequential order
- **critical** : ensure that the following block be executed one thread at the same time
- **atomic** : ensure atomic operation on following variable
- **master** : ensure that the following block be executed only bay master thread
- **locks**

OpenMP

TASK since OpenMP 3

TASK

#pragma omp task [*clause* [*clause*]. . .]
{
// BLOCK
}

- the current thread creates a task with the following block
- the task is added to a pool of tasks

Clauses :

- **if(expr)** : the task is executed by the current thread if expr=true
- **final(expr)** : sub tasks ar integrated to the current task if expr=true
- **untied** : any thread can executed if task is suspended

Synchronization :

- **#pragma omp taskwait** : define a barrier to wait that all sub tasks are achieved

OpenMP
Runtime library

Library :

void omp_set_num_threads(int n) void omp_set_dynamic(int bool) void omp_set_nested(int bool) void omp_set_max_active_levels(int n) void omp_set_schedule(omp_sched_t type, int chunk)	
int omp_get_num_threads() int omp_get_dynamic() int omp_get_nested() int ompget_max_active_levels() void omp_get_schedule(omp_sched_t* type, int* chunk)	
int omp_get_thread_num() int omp_get_num_procs() int omp_in_parallel() int omp_in_final()	

OpenMP
Environment variable

Environment variables :

OMP_NUM_THREADS	number of threads for parallel region
OMP_SCHEDULE	define schedule policy
OMP_DYNAMIC	true or false enable runtime adjust num of threads
OMP_NESTED	true or false to activate nested parallelism

OpenMP
Concurrency management

Two types of locks :

- **omp_lock_t**
- **omp_nest_lock_t**

Library functions :

void omp_init_lock(omp_lock_t* l)	initialize a lock l
void omp_destroy_lock(omp_lock_t* l)	destroy the lock l
void omp_set_lock(omp_lock_t* l)	lock the lock l
void omp_unset_lock(omp_lock_t* l)	unlock the lock l
void omp_test_lock(omp_lock_t* l)	try to lock l, return true if succeeded

Outline

Task Programming with TBB

Introduction to TBB

- Intel Threading Building Blocks
- enabling parallelism in C++ applications and libraries
- provides :
  - generic parallel algorithms,
  - concurrent containers,
  - support for dependency and data flow graphs,
  - thread local storage,
  - a work-stealing task scheduler for task based programming,
  - synchronization primitives,
  - a scalable memory allocator,
  - ...

Task Programming with TBB

Generic Parallel Algorithms

- parallel_for: map
- parallel_reduce, parallel_scan: reduce, scan
- parallel_do: workpile
- parallel_pipeline: pipeline
- parallel_invoke, task_group: fork-join
- flow_graph: plumbing for reactive and streaming apps

Task Programming with TBB

Principle : Applying generic algorithm C++

Serial program

```
void SerialApplyFoo( float a[],
                     size_t n )
{
    for( size_t i=0; i!=n; ++i )
        Foo(a[i]);
}
```

Parallel program

```
include "tbb/tbb.h"
using namespace tbb;
void ParallelApplyFoo( float a[],
                       size_t n )
{
    parallel_for( size_t(0), n,
                  [&]( size_t i )
                  {
                      Foo(a[i]);
                  }
            );
}
```

Task Programming with TBB

Principle : Block Range concepts

Range 1D

```
{
  using namespace tbb;
  parallel_for(blocked_range<int>(0,nrows),
               [&](blocked_range<int> const r)
               {
                 for(auto irow=r.begin();irow<r.end();++irow)
                   for(int j=0;j<ncols;++j){
                   ...;
                   }
               } ) ;
}
```

Task Programming with TBB

Principle : Block Range concepts

Range 2D

```
{
  using namespace tbb;
  parallel_for(blocked_range2d<int>(0,nrows,0,ncols),
               [&](blocked_range2d<int> const r)
               {
                 for(auto i=r.rows().begin();i<r.rows().end();++i)
                   for(auto j=r.cols().begin();j<r.cols().end();++j){
                     ...;
                   }
               } ) ;
}
```

Task Programming with TBB

Principle

Serial containers

```
extern std::queue<T> MySerialQueue;
T item;
if( !MySerialQueue.empty() ) {
   item = MySerialQueue.front();
   MySerialQueue.pop_front();
      ... process item...
}
```

Concurrent containers

```
include "tbb/tbb.h"
using namespace tbb;
extern concurrent_queue<T>
MyQueue;
T item;
if( MyQueue.try_pop(item) ) {
   ...process item...
}
```

Task Programming with TBB

Principle

Task group

```
include "tbb/task_group.h"
using namespace tbb;
int Fib(int n) {    if( n<2 ) {
      return n;
   } else {
      int x, y;
      task_group g;
      g.run([&]{x=Fib(n-1);}); // spawn a task
      g.run([&]{y=Fib(n-2);}); // spawn another task
      g.wait(); // wait for both tasks to complete
      return x+y;
   } }
```

Task Programming with TBB

Principle

Synchronization

```
Node* FreeList;
typedef tbb::spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;
Node* AllocateNode() {
   Node* n;
   {
      FreeListMutexType::scoped_lock lock(FreeListMutex);
      n = FreeList;
      if( n ) FreeList = n->next;
   }
   if( !n ) n = new Node();
   return n;
}
```

Task Programming with TBB

Principle

Synchronization

```
void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
```

Task Programming with TBB

Principle

Atomic operation

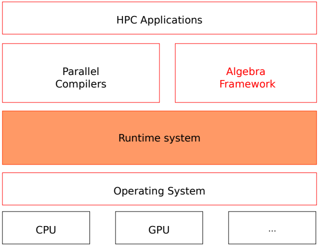
```
tbb::atomic<T> x ;
y = x ; //read the value of x
x= expr ; //write the value of x, and return it
x.fetch_and_store(y) ; //do x=y and return the old value of x
x.fetch_and_add(y) ; //do x+=y and return the old value of x
x.compare_and_swap(y,z) ; //if x equals z, then do x=y. In either case, return old value of x.
```

Outline

Programming and run time system

State of art

- Within libraries
  - Quark scheduler
  - TBLAS data management
- Within compiling environments:
  - TBB, OpenMP,...
  - HMPP, PGI, OpenACC,...
- With emerging standards:
  - OpenCL, OpenACC
- Research Runtime systems:
  - Charm++ (Urbana, UIUC)
  - StarSS, OmpSs (Barcelona, BSC)
  - StarPU (INRIA Bordeaux)
  - HPX (indiana university)



Parallelism at instriction level

Introduction to SIMD

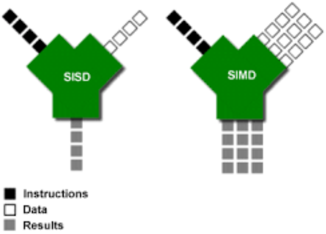


Figure: SIMD principle

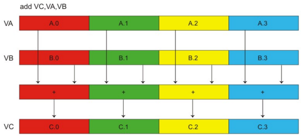


Figure: SIMD ADD operation

Outline

Parallelism at instruction level

SIMD with OpenMP

OPENMP directive

loop : **#pragma omp simd**
function : **#pragma omp declare simd**
Parameters :

- **aligned(list[:])**
- **collapse(n)**
- **reduction(op-id:list)**
- **safelen(length)**
- **simdlen(length)**

Loop

```
for(int i=0;i<n;++i)
   y[i] = 2.0 * x[i];
double sum = 0. ;
for(int i=0;i<n;++i)
   sum += 2.0 * x[i];
```

Vectorized Loop

```
#pragma omp simd
for(int i=0;i<n;++i)
   y[i] = 2.0 * x[i];
double sum = 0.;
#pragma omp simd reduction(+:sum)
for(int i=0;i<n;++i)
```

TP

Introduction to Task programming

- TP 1 : Hello word
  - using std::thread ;
  - using OpenMP loops ;
  - using OpenMP tasks ;
  - using OpenMP TBB ;
- TP 2 : Matrix Vector product
  - Dense Matrix format ;
  - Sparse Matrix format ;
- TP 3 : LU algorithm
  - Parallel WaveFront pattern
- TP 4 : Image processing
  - Median filter ;
  - Connected Component Labelization