Parallel Programming for Exa-Scale Computing
Nicolas Maillard
nicolas@inf.ufrgs.br
ASM 2011, Porto Alegre

Outline
- The GPPD,
- ExaScale: Challenges,
- Parallel Programming Landscape,
- Task parallelism: what, why, how?

Exa-Flops/s == 10^{18} Challenges
- Heterogeneity
  - GPUs
- Fault-Tolerance,
  - Global state
- Scheduling/mapping,
  - Dynamicity
- Bandwidth/data-transfer,
  - Locality
- IO,
- Power supply,
- ...

OpenMP: default for Shared-Memory Systems

Sequential code
double res[10000];
for (i=0 ; i<10000; i++)
  compute(res[i]);

Parallel Open-MP code
double res[10000];
#pragmaomp parallel for
for (i=0 ; i<10000; i++)
  compute(res[i]);

Data Parallelism
- Simple idea
  - Define your data-structures (e.g. arrays, trees...);
  - Decide how you distribute them among your CPUs;
  - Run the computation in parallel on each local piece of data.
    With few dependencies, simple and efficient.
Example: CUDA for GPUs

```c
void MatrixMul(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd; // Vectors that will be processed in parallel.
    dim3 dimGrid(1, 1);
    dim3 dimBlock(Width, Width);
    // Call a function to be run by each CUDA thread on its own
    // piece of data:
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
}

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Global ID's
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x < Width && y < Width)
    {
        double valor;
        if (global threadID is 0)
        {
            valor = 3.14;
        }
        Md[x] = Md[x] * Nd[y] + valor;
    }

    // Send/Receive
    MPI_Sendrecv(Md, Width, MPI_DOUBLE, 0, 0, Pd, 0, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

Message Passing Interface for Distributed Memory

```c
void main() { ...
    MPI_Status status;
    double valor;
    if (r==0) {
        // Processor 0 sends a message to 1
        valor = 3.14;
        MPI_Sendrecv(MD, P, MPI_DOUBLE, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
    } else {
        // Processor 1 receives a message from 0
        MPI_Sendrecv(MD, P, MPI_DOUBLE, 0, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    }
}
```

A MPI parallel program

*p processes interact through message passing.*

Scalability of MPI

- **MPI on a Million Processors?**
  - Limitations in the API (e.g.: some calls may need $O(p)$ arguments),
  - Limitations in the implementation
    - Mapping rank/pid in a communicator
    - How to enable fault-tolerance in MPI?
Anyway...

• As always in Parallel Programming, the conclusion is that:
  – There is still no universal, ideal API for parallel programming (at whatever scale),
  – You need to redesign your program/algorithm if you want to scale to the Exa.
    • No all2all,
    • 3D decomposition of the data,
    • ...

What are tasks?

• Task: set of sequentially ordered instructions.
• Tasks have an input/output, can synch(), can communicate, can be stored, can be run.
• Fine granularity
  – The Task is a (virtual) entity which MAY be run in parallel by resources from the OS/HW. Or not.

Why do you want tasks?

• “This is not the way one programs with (MPI|OpenMP|Pthreads|CUDA...!”
  – Okay – but MPI is what you have! Is it what you want?

Parallel Programming Model

• Programming model: how do you want to write down your algorithms?
  – You want to specify/program your algorithm with as much parallelism as possible, independently of any hw constraint.
• In the sequential world:
  – Do you care about the mapping of the vars to registers/stack/heap?
  – Do you split your instructions into functions based on their runtime?
  – So why do you want to do it in parallel?
How do you “think parallel”?

Sequential or parallel?

Tasks and Recursivity
Scheduling tasks

- PRAM theory and Brent Theorem: from a fine-grained highly parallel program, without HW limit, you can deduce an efficient (i.e. with good speed-up) program.
- $T_p(n) = T_w + W_{par} / p$
- Yet, you need an efficient scheduler.
- In practice: Work Stealing serves.

Tasks with CILK


```c
void ParallelApplyFoo(float a[], int n)
{
  // Parallel loop
  for (int i=0; i<n; i++)
    ApplyFoo(a[i]);
}
```

OpenMP

- Traditional OpenMP programming is based on **loop parallelism**.
  - The task is an iteration.

```c
function soma(integer n, real v[:])
  real v[:]
  real res, res1, res2
  if (n eq 1) then res=v(1)
  else
    temp parallel for reduce(res,+)
      do i=1, n
        res = res + v(i)
      end do
    soma = res
  return
end
```

Tasks in OpenMP

- Traditional OpenMP programming is based on **loop parallelism**.
  - The task is an iteration.
- OpenMP3 (mid 2008) has introduced Task parallelism
  - Heritage from Cilk

```c
 function soma(integer n, real v[:])
   real v[:]
   real res, res1, res2
   if (n eq 1) then res=v(1)
   else
     spawn task
     res = soma(v(2), v(n/2))
     end task
     res1 = soma(v(2), v(n/2:n))
     end task
     res2 = soma(res1, res2)
     soma = res
   return
end
```

Intel TBB

- [http://www.threadingbuildingblocks.org](http://www.threadingbuildingblocks.org)

Intel’s proposal for Multicore programming.
  - Based on C++ STL/Containers
    - “C++lear” syntax
  - The tasks are the elements in the container
    - You can “iterate” on the tasks in parallel
    - You can apply algorithms to the tasks.

Example TBB

```c
// Sequential loop
void SerialApplyFoo(float a[], int n)
{
  for (int i=0; i<n; i++)
    Foo(a[i]);
}
```

```c
// Parallel loop
class ApplyFoo
{
  void operator()()
  {
    const blocked_range<int>& r = const_cast
    (float* my_a, int n)
    void operator()()
    {
      for (int i=0; i<n; i++)
        for (int i=0; i<n; i++)
          if (i < n)
            h = h + a[i];
      } // Constructor
  }
}
```
Tasks with Kaapi

A French project - [http://kaapi.gforge.inria.fr/](http://kaapi.gforge.inria.fr/)

C++ library that provides an API for parallel programming based on tasks.

A shared global address space

- You create objects inside it with the keyword “shared”
- A task is a call to a function, prefixed by “fork”
  - Just like Unix / Cilk
  - Tasks communicate through objects shared.
  - Tasks specify the access mode to shared objects (read/write)
- The Kaapi runtime builds the data-flow graph and uses it to schedule the tasks.

A simple example

```cpp
struct Fibonacci {
  void operator()(int n, a1::Shared_w<int> result)
  {
    if (n < 2) result.write(n);
    else
      a1::Shared<int> subresult1;
      a1::Shared<int> subresult2;
      a1::Fork<Fibonacci>()(n-1, subresult1);
      a1::Fork<Fibonacci>()(n-2, subresult2);
      a1::Fork<Sum>()(result, subresult1, subresult2);
  }
}

struct Sum {
  void operator()(a1::Shared_w<int> result, a1::Shared_r<int> sr1, a1::Shared_r<int> sr2)
  {
    result.write(sr1.read() + sr2.read());
  }
}
```

Tasks & Heterogeneous Parallel Programming

- Adaptive Work Stealing already uses 2 algorithms
  - 1 sequential, 1 parallel.
- Why not using 2 different implementations (methods) to run on a container, e.g. one for CPU and one for GPU?
  - The merge() method handles the different address spaces.
- This has been done partially by B. Raffin and E. Hermann [EGPGV 09].
- J. Lima’PhD.

Tasks with MPI?

[Image of a person using a laptop]
The MPI task

- MPI defines tasks that:
  - Have their own address space,
  - Communicate with other tasks through messages.
  - Usually all are launched at the start of the program.
- The mapping "MPI task" / O.S. is not specified.
  - Usually, 1 task == 1 heavy process (O.S. view);
  - MPI 2 has somewhat reinforced this common understanding
  - Some MPI Distributions use threads (MPICH);
  - A-MPI defines an abstract task (Urbana Champaign)

Dynamic Tasks in MPI: D&C and Spawn

- Program with Divide & Conquer techniques.
- Use MPI_Comm_spawn (recursively) create new tasks.
  - 1 task == 1 (MPI) process.
- Make sure that the children tasks may communicate with their parent
  - Have the parent send the children input data, and then block.
  - Have the children send their results back to the parent.
- This implies very large-grained parallelism, but at least:
  - You can benefit from dynamic resources.
  - You can improve the load balance.

Parallel runtime of Eratosthenes' Sieve

Cluster usage with Rigid jobs + malleable, MPI programs

MPI tasks: processes vs. threads

- João Lima’s Master
- Modification of MPICH (C++ API).
- Transparent to the user;
- The spawned tasks may be run by processes or threads.
  - Does not change the Send/Recv semantics.

Mergesort of 3M numbers.
Validation with “Real-World” apps

- OLAM - Ocean-Land-Atmosphere Model.
- Uses finite elements

Conclusions

- To reach the ExaFlops/s, new tools are required:
  - New hardware, better I/O solutions, better interfaces with the user (visualization...)
  - A parallel programming model that decouples the parallelism from the hardware.

Task-based parallelism may be one way.
- There are solutions for shared-mem. systems (TBB, OpenMP...)
- The support for distributed memory is still an on-going research (Kaapi, Charm++...)

Other ideas to go further

- It is probably utopic to try and devise one more API for parallel programming.
- Automatic generation of code
  - Different from Compiling: support to the programmer.
  - Source2source compiling, interaction with the programmer.
- Auto-Tuning