Panorama de la programmation parallèle dans le HPC

F. Bodin
Irisa / Université de Rennes

1

Introduction

- Parallel programming languages are at the core of software development
  - Ubiquitous parallel machines
  - Legacy codes extremely important
- Languages and APIs are always based on some hardware characteristics assumptions
  - Deep changes in HW raises language / APIs questions
- Long term trend due to the *power wall*
  - Requires reviewing the design of application codes
- HPC reaching more technical fields
  - e.g. high frequency trading, …

2

History Repeating Itself

In 1994, after pioneer companies *Thinking Machines* and *Kendall Square Research* went Chapter 11, Ken Kennedy wrote : ***Is Parallel Computing Dead?***

[…] So I assert that parallel computing is not dead, it is simply suffering through a period of growing pains. ***We are coming to realize that parallel computing is really a software problem***. Therefore, we should not be too distressed when a company drops out of the crowded hardware market. ***The industry needs software and standards to make writing parallel applications easier. If it gets them, the whole industry will grow, creating more business for everyone.***

3

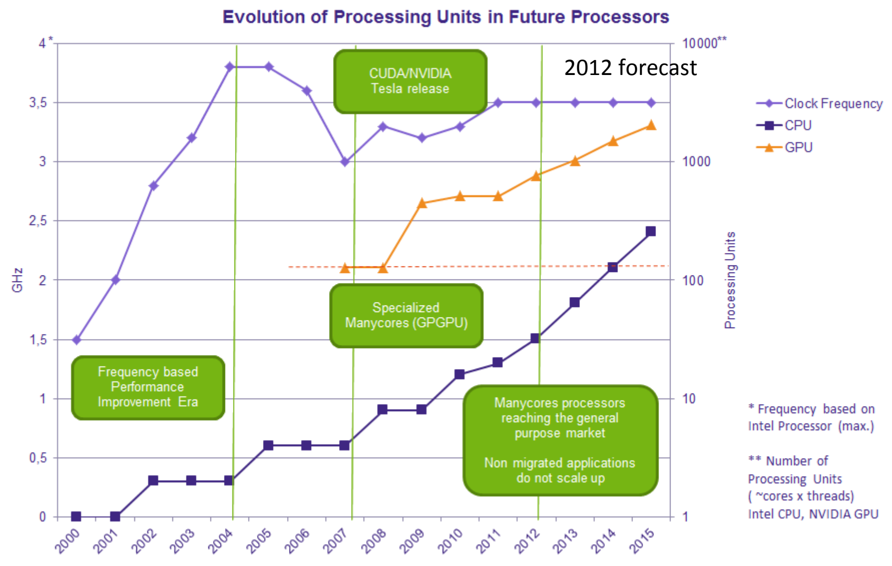
Outline

- Parallel programming in context
- General purpose parallel languages and APIs
- The response to heterogeneity (e.g. GPGPU)
- Languages / APIs still to emerge

4

2014: Intel Haswell-EP 18x2HTx2sockets = 72

Intel Ivy Bridge-EX 15 cores/sockets, 2threads/core, max 8 sockets = 240

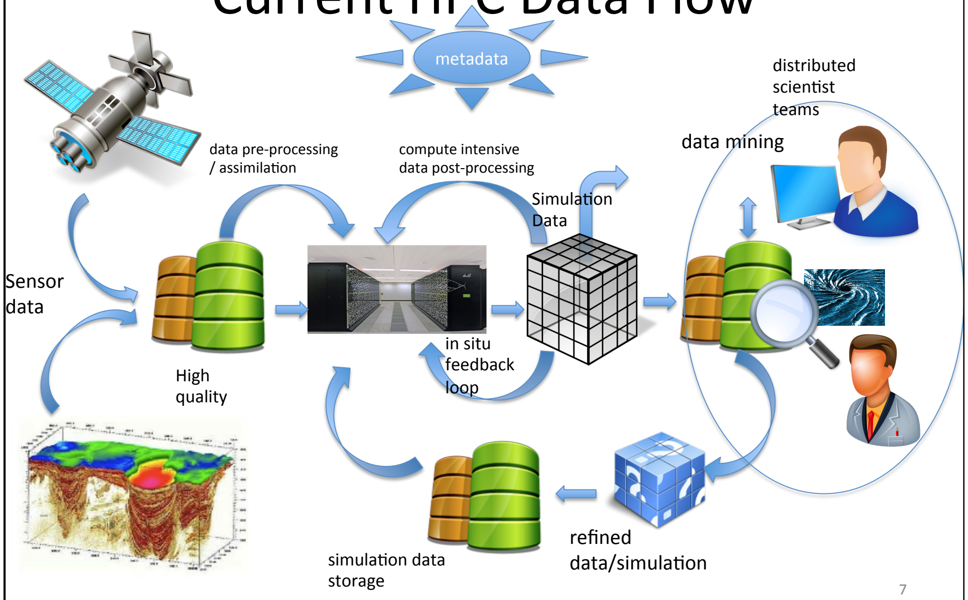


2014: Nvidia K40 → 2880 CUDA cores

Parallel Programming in Context

- Part of a large ecosystem
  - Programming languages and APIs a tiny part of the equations
- Scalability, a primary concern
  - Ensuring scalability on a wide range of systems
- Economical constraints driving engineering consideration
  - e.g. code validation issue can be a roadblock

Current HPC Data Flow



7

Code Technical Main Matters

- Code validation
- Surviving at least 4 generations of (very different) machines
- Sophisticated runtime techniques needed to optimize resources and energy consumption
- Data structure organization
- IO sub-systems / data management
- Application development process

8

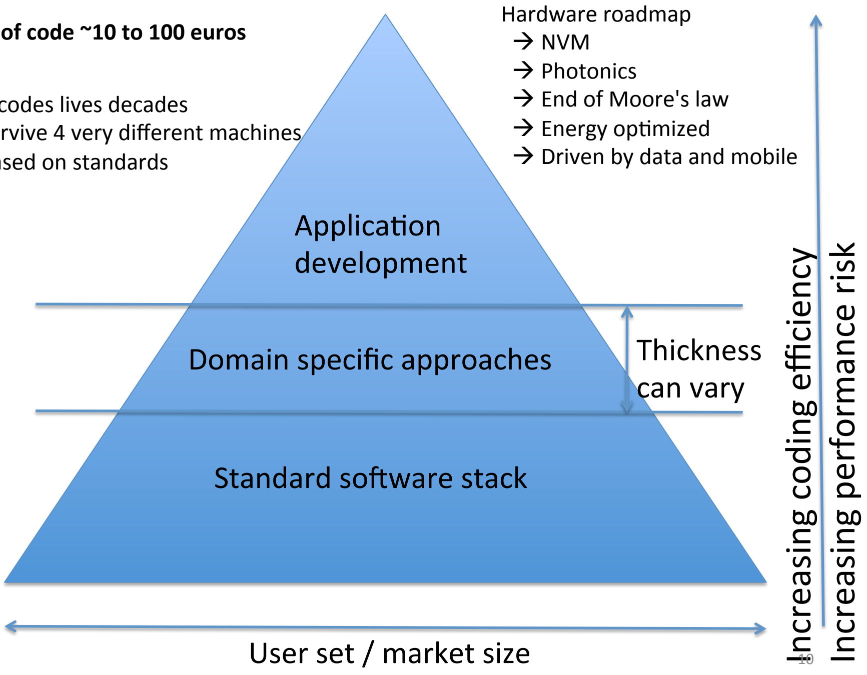
Code Economical Factors

- Cost of development
  - How much for a line of code?
- Code exploitation duration
  - How long before a major rewrite of the code?
- Support and maintenance
  - Am I mastering all issues?
- Performance
  - Do I get efficient use of hardware, what are the tradeoffs?
- Technical risk
  - Roadblocks ahead? How costly is it to fix it?

A line of code ~10 to 100 euros

Large codes lives decades
→ survive 4 very different machines
→ based on standards

Hardware roadmap
→ NVM
→ Photonics
→ End of Moore's law
→ Energy optimized
→ Driven by data and mobile



Protecting Software Investment

- Mastering risks if the first engineering consideration
- Standards "a must have" for the long run
  - Libraries and tools must be included here
- Conservatism is the main driver
  - Too much money at stake
  - Precedence of functionalities over better technology
- Adopting new "languages" has a major impact in well established practices
  - Not only coding but also the deployment process and training

11

GP Parallel Languages and APIs

12

GP Parallel Languages and APIs

- Main parallel programming models
  - Tasks based ~ shared memories
  - Message passing ~ distributed memories
- MPI (~distributed memories) and OpenMP (~shared memories)
  - Available on many platforms
  - Available for many languages
- Evolving (old) standards
  - Many implementations and supporting companies
  - Characteristics well understood
- Designed decades ago

13

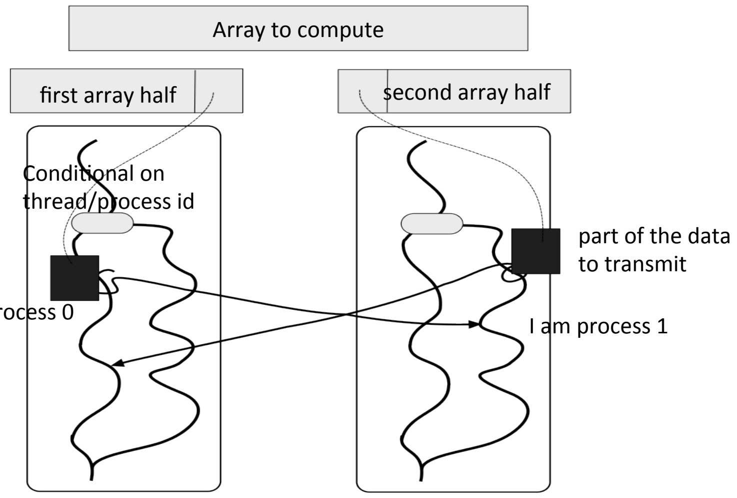
Message Passing Interface (MPI)

- In this model the tasks send and receive messages in a synchronous or asynchronous way
  - Explicit control of all inter-processor communications
- The dominant form of large scale parallelism in scientific computing
  - No thread creation overhead
  - Can run on shared or distributed memory architecture
- Mainly based on the Single Program Multiple Data (SPMD) model

14

SPMD and Message Passing

Data parallel approach usually Implemented this way



All processes/threads execute the same program/code

Single Program Multiple Data

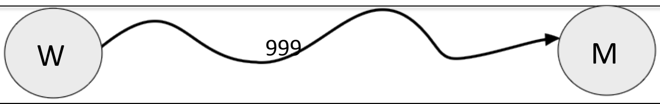
A MPI Example (1)

```
int main(int argc, char *argv[]){
  int   numtasks, taskid, len;
  char hostname[MPI_MAX_PROCESSOR_NAME];
  MPI_Status status;
  MPI_Request mySend;
  int tab[1] ={0};
  int label1=1;
  //just starting the processes
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  switch(taskid){
     ....
  }
  MPI_Finalize();
}
```

```
mpirun -np 2 messages.exe
Task 0 on hmpp.irisa.fr!
MASTER: Number of MPI tasks is: 2
master receiving value 999
Task 1 on hmpp.irisa.fr!
worker asynchronously sending value 999
```

A MPI Example (2)

```
switch(taskid){
 case MASTER:
   MPI_Recv(tab, 1, MPI_INT, WORKER1,
            label1, MPI_COMM_WORLD, &status);
   ...
   break;
 case WORKER1:
   tab[0] = 999;
   MPI_Isend(tab, 1, MPI_INT, MASTER,
   label1, MPI_COMM_WORLD,&mySend);
   break;
   . . .
 }
```



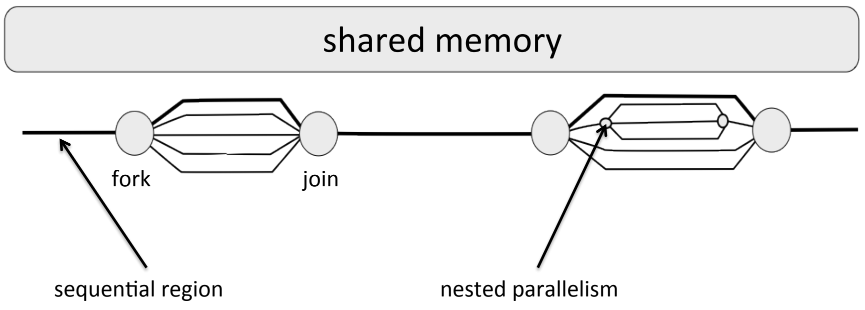
17

OpenMP Approach

- OpenMP is an API for writing multi-threaded applications
  - A set of directives
  - Runtime routines
  - Environment variables
  - Target shared memory architectures
  - Promote incremental parallelization
  - Available on many platforms/compilers
  - Available for C/C++ and Fortran
- Definition started in the 80s'
- Program is decorated to allow the compiler to generate parallel code
  - Mainly fine grain parallelism, mostly loop level

18

OpenMP Parallel Model

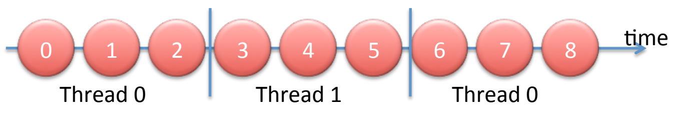


- The whole the programming environment assumes some level of threads management
  - Gang scheduling
  - Thread / data affinity / migration

OpenMP Example: Parallel Loop

- Loop iterations are distributed to the treads according to a scheduling policy (here static)

```
tstart = wallclock();
#pragma omp parallel for private(j) schedule(static,3)
  for (i=0;i<N;i++){
    for (j=0;j<N;j++){
      M[i][j] = X[i]*Y[j];
    }
  }
tend = wallclock();
```



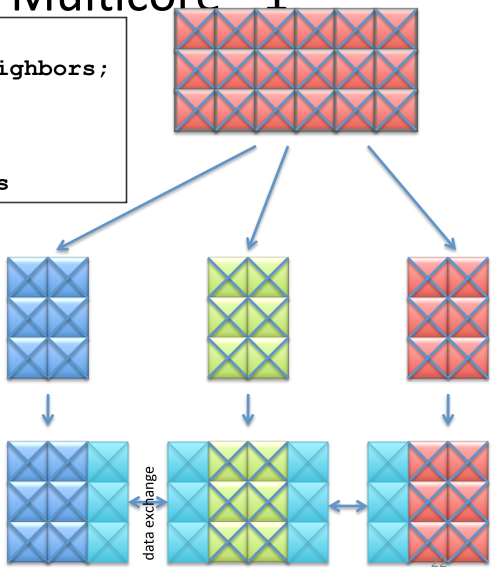
A Slowly Moving Landscape

- Many HPC parallel codes are implemented using MPI, with a process per CPU core
  - **This strategy is unable to efficiently exploit current multicore memory systems**
- Mixing OpenMP and MPI is becoming more popular
  - Code tuning getting more and more complex
- Vectorization importance growing
  - Still no satisfactory answers

21

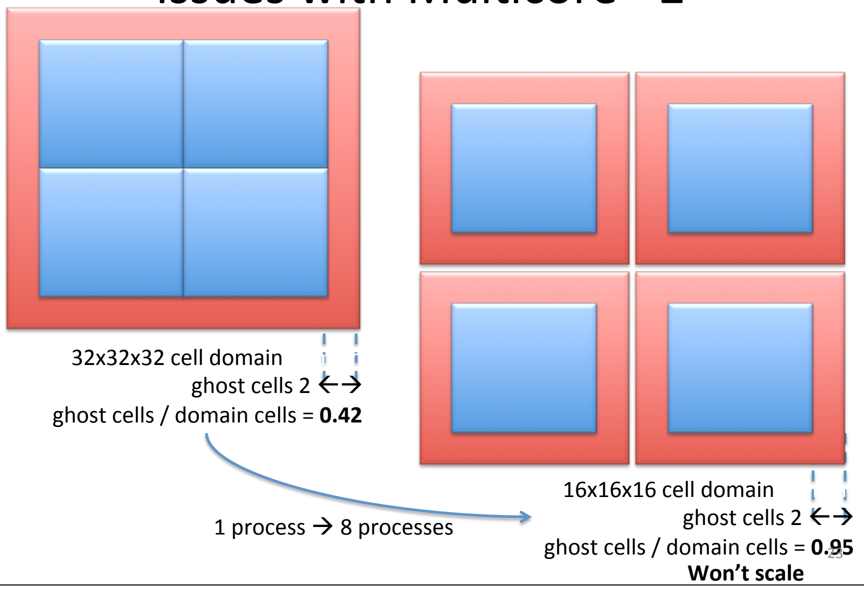
Domain Decomposition Parallelism Issues with Multicore - 1

```
forall cells of PE
   read properties, state, neighbors;
for time = 1 to end
  forall cells of PE
      compute next step
  update neighbors ghost cells
```



1. Domain decomposition
2. Add ghost cells for communications

Domain Decomposition Parallelism Issues with Multicore - 2



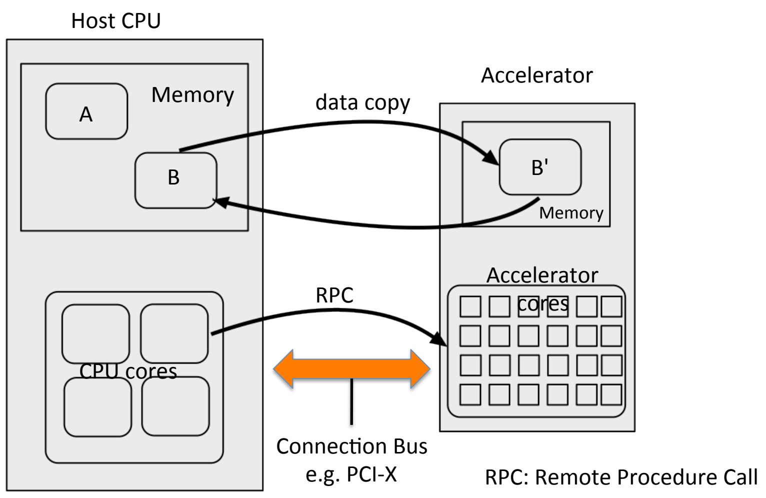
The Response to Heterogeneity

The Response to Heterogeneity

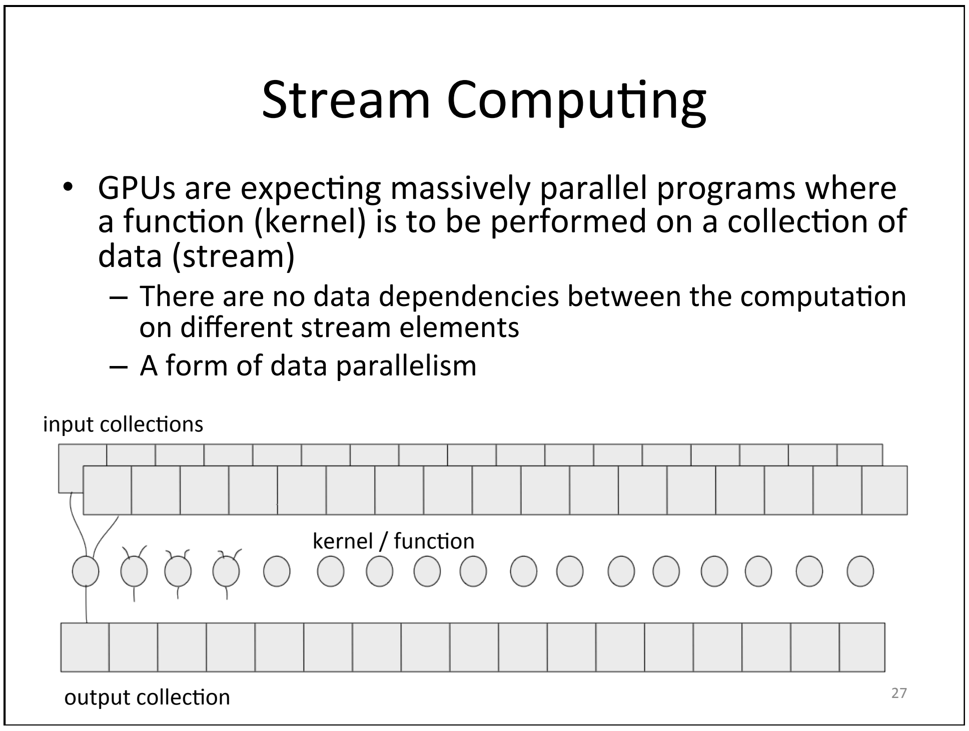
- General-Purpose computation on Graphics Processing Units
  - Become very popular with Nvidia releasing CUDA in 2007
  - GPGPU exploits GPUs massively parallel computing architectures as accelerators
- Heterogeneity is a response to the power wall
- Pushes the issues to the programmers

25

Accelerator Offloading Model

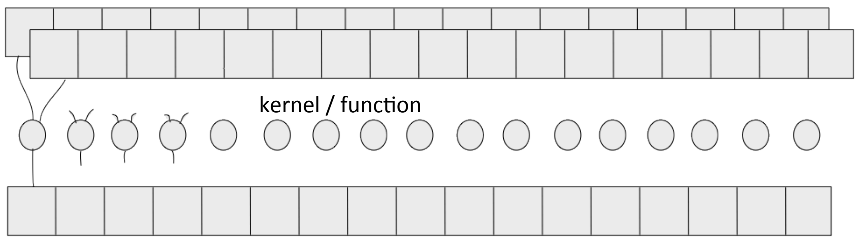


26

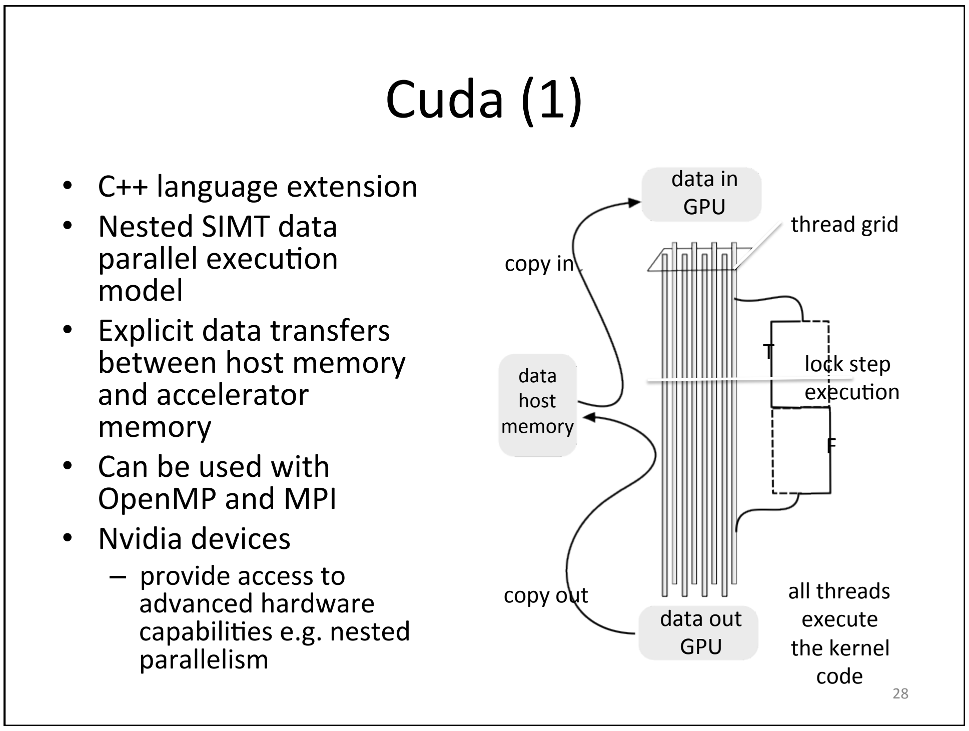


Stream Computing

- GPUs are expecting massively parallel programs where a function (kernel) is to be performed on a collection of data (stream)
  - There are no data dependencies between the computation on different stream elements
  - A form of data parallelism

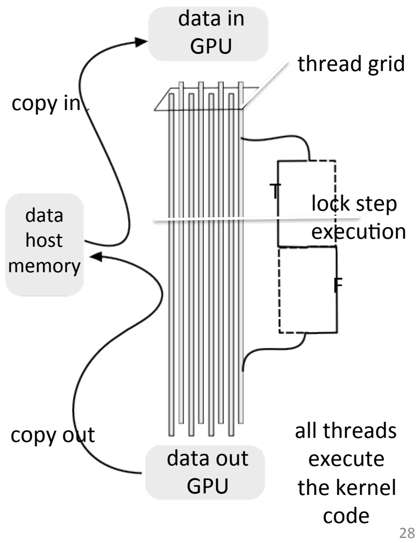


27



Cuda (1)

- C++ language extension
- Nested SIMT data parallel execution model
- Explicit data transfers between host memory and accelerator memory
- Can be used with OpenMP and MPI
- Nvidia devices
  - provide access to advanced hardware capabilities e.g. nested parallelism



28

OpenCL (1)

- C language extension
  - Similar parallel model to CUDA
  - Available on many devices, HPC and mobile
  - OpenCL vectorizing friendliness → Intel Xeon Phi implementation
- Underlying assumptions
  - same as Cuda
- Some issues not covered
  - Similar to Cuda + C++, Fortran
  - Remote data accesses
  - Nested parallelism
  - **Not all hardware capabilities available**
    - Standard specification latency

29

OpenACC (1)

- Fortran, C, (C++) directives set
- Mainly target legacy codes
- Can be mixed with regular OpenMP and MPI

```
float A[n];

#pragma acc data create(A)
{
  #pragma acc kernels present(A)
  {
    for(i=0; i < n; i++) {
      A[i] = B[n - i];
    }
  }
  …
  init(C)
  …
  #pragma acc kernels present(A)
  {
    for(i=0; i < n; i++) {
      C[i] += A[i] * alpha;
    }
  }
}
```

30

Synthesis (1)

- Current state of the art reflect contradictory goals
  - Power efficiency
  - Providing abstract parallel programming API/lang.
- Stable, widely adopted, API/lang. difficult to propose before hardware convergence
  - Directives designed to fill the gap
  - Shared / global address space is a must have to simplify accelerator uses
- Main roadblock is the algorithm / numerical model in most cases, not the programming API/language

31

Languages / APIs Still to Emerge

32

Languages / APIs Still to Emerge

- Partitioned Global Address Space
  - Simplifying remote data access programming
- Interpreted languages
  - Simplifying access
- Domain specific Languages
  - Abstracting the issues

Partitioned Global Address Space (PGAS)

- Provide a virtual partitioned global namespace
  - Threads can directly access remote data
- Data is either local or global
  - Promote data locality oriented programming
- PGAS Languages: UPC, Co-Array Fortran, Titanium, etc.

PGAS UPC Example

```
shared int hits;
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
   upc_lock_t *hit_lock = upc_all_lock_alloc();
   int trials = atoi(argv[1]);
   my_trials = (trials + THREADS - 1)/THREADS;
   srand(MYTHREAD*17);
   for (i=0; i < my_trials; i++) my_hits += hit();
   upc_lock(hit_lock);
     hits += my_hits;
   upc_unlock(hit_lock);
   upc_barrier;
   if (MYTHREAD == 0)
      printf("PI: %f", 4.0*hits/trials);
}
```

Example from “Beyond UPC”, Kathy Yelick, UC Berkeley

35

PGAS Strengths and Weaknesses

- Strengths
  - Data locality oriented
  - One sided communications
  - Data management and synchronization part of the language
  - Smaller code size than MPI
- Weaknesses
  - SPMD parallel programming model
  - Adoption (CAF started in 1998, 2008 standard)
  - Parallel libraries

36

Interpreted Languages

- New ways for access to HPC
  - High level approach, dynamic
- For instance Julia (MIT)
  - Automatic memory mngt
  - One side data access
  - Remote function call
  - Meta-programming
  - Dense, sparse matrices
  - Plotting capabilities
  - JIT for performance
  - Cluster-wise
  - …

```
function buildVector()
   vec=[0:0.01:10];
end
x = buildVector();
y = 0.0*x;
for n=1:length(x)
   y[n] = cos(x[n]);
end
plot(x,y);
r1 = remotecall
     (2,buildVector);
fetch(r1)
```

Example of Julia code

37

Domain Specific Approaches

- Aims at abstracting implementation details
  - High level specification of computation
  - Code generation for parallel and vector

$$
\begin{aligned}
Dirac = {} & I_{L\otimes C\otimes S} \\
& + 2 * i * \kappa * \mu * I_{L\otimes C} \otimes \gamma_5 \\
& + \kappa * \sum_{d\in D} ((J_L^{-d} \otimes I_C) * \bigoplus_{s\in L} U(d)[s]) \otimes (I_S + \gamma[d]) \\
& + \kappa * \sum_{d\in D} ((J_L^{d} \otimes I_C) * \bigoplus_{s\in L} U(-d)[s]) \otimes (I_S - \gamma[d])
\end{aligned}
$$

Definition of Dirac matrix on a Lattice L in QIRAL

- Examples
  - DCT generation (e.g Spiral)
  - Stencil code generation (Patus)
  - Code generation for quantum chemistry tensor product
  - Quantum chromodynamics (QCD), subset of LaTeX (Qiral)

38

DSL Pros and Cons

- Pros
  - High abstraction level, domain specific
  - High level semantic → powerful code generation
- Cons
  - Integration in legacy / GP codes
  - Market size / development cost
  - Sustainability, support and maintenance
  - Coverage

39

Perspectives: A Race for Adoption

- PGAS have been there for a long time
  - e.g. Co-array Fortran, 2008 standard
- Interpreted languages growing fast
  - e.g. Python, Julia successes
  - New HPC users (?)
- DSL
  - Economical viability
  - Integration in development process
  - Necessary to hide complexity

40

Conclusion

- Hardware is still a moving target
  - New technologies such as NVM and photonics likely to change the landscape
- Homogeneous programming is at an end
  - More asynchronous parallel tasks to deal with the growth of the number of cores
- Code architectures more important than languages
  - But but choices can have expensive consequences
- Algorithms more important than architectures
  - Exhibiting parallelism is the first roadblock
  - Hierarchy of parallelisms

41

Some Readings - 1

- **Non-Volatile Memory (NVM) technology for Exascale computing - A position paper**http://www.hpcmagazine.com/state-of-the-art/non-volatile-memory-nvm-technology-for-exascale-computing-a-position-paper/
- **Processor evolution: what to prepare application codes for?** http://www.hpcmagazine.com/state-of-the-art/processor-evolution-what-to-prepare-application-codes-for/
- **Multicore Programming Practices, Multicore association, 2013** http://www.multicore-association.org/workgroup/mpp.php
- **C. Lin, L. Snyder, “Principles of Parallel Computing”, Pearson, 2008**

42

Some Readings - 2

- **Keynotes on HPC Languages, Lyon, 2013**
  - http://labexcompilation.ens-lyon.fr/hpc-languages/
- **Julia**
  - http://julialang.org

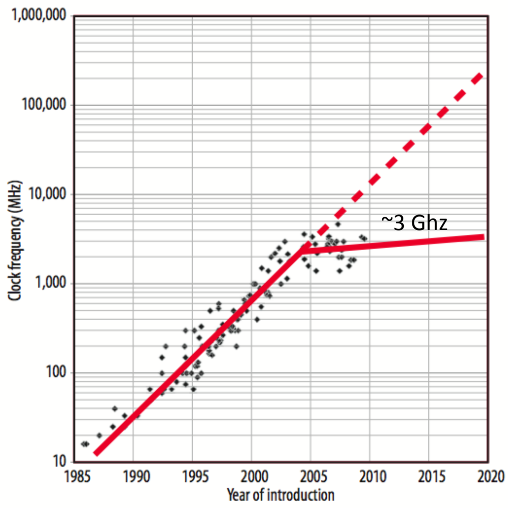
43

BACKUP SLIDES

44

Historical growth and forecast in single-processor frequency

- Based on the ITRS roadmap*
- A dashed line represents expectations if single-processor performance had continued its historical trend



*http://www.itrs.net

45

Software Development Cost Estimating Guidebook,
Software Technology Support Center Cost Analysis Group, DoD, October 2010

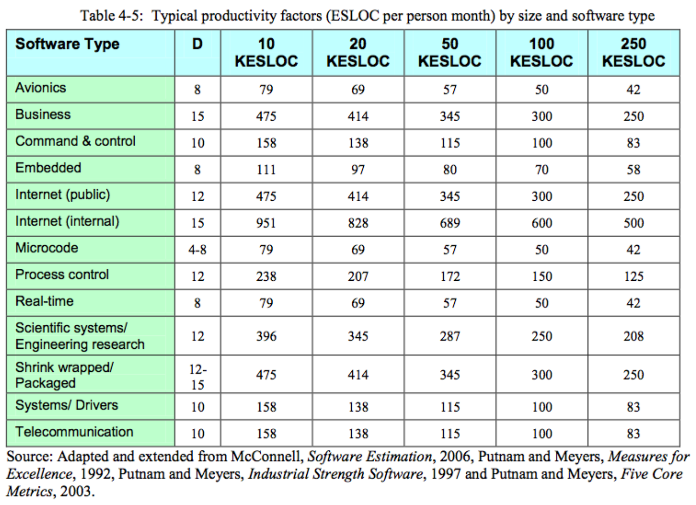


Table 4-5: Typical productivity factors (ESLOC per person month) by size and software type

Software Type	D	10 KESLOC	20 KESLOC	50 KESLOC	100 KESLOC	250 KESLOC
Avionics	8	79	69	57	50	42
Business	15	475	414	345	300	250
Command & control	10	158	138	115	100	83
Embedded	8	111	97	80	70	58
Internet (public)	12	475	414	345	300	250
Internet (internal)	15	951	828	689	600	500
Microcode	4-8	79	69	57	50	42
Process control	12	238	207	172	150	125
Real-time	8	79	69	57	50	42
Scientific systems/ Engineering research	12	396	345	287	250	208
Shrink wrapped/ Packaged	12-15	475	414	345	300	250
Systems/ Drivers	10	158	138	115	100	83
Telecommunication	10	158	138	115	100	83

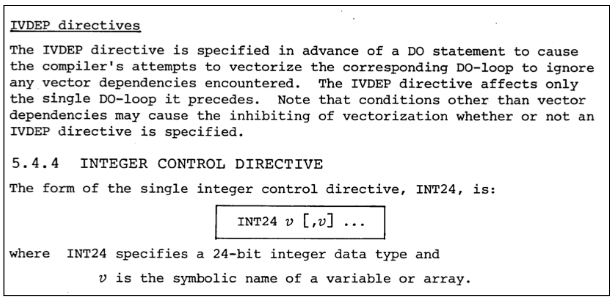
Source: Adapted and extended from McConnell, *Software Estimation*, 2006, Putnam and Meyers, *Measures for Excellence*, 1992, Putnam and Meyers, *Industrial Strength Software*, 1997 and Putnam and Meyers, *Five Core Metrics*, 2003.

D: complexity measure, higher more complex
KESLOC: thousand effective source lines of code

46

Pragma / Directives

- Pragma / directives are comments in program taken into account by the compiler
- Specify data or commands to drive the code generation process
- Can also be ignored



IVDEP directives

The IVDEP directive is specified in advance of a DO statement to cause the compiler's attempts to vectorize the corresponding DO-loop to ignore any vector dependencies encountered. The IVDEP directive affects only the single DO-loop it precedes. Note that conditions other than vector dependencies may cause the inhibiting of vectorization whether or not an IVDEP directive is specified.

5.4.4 INTEGER CONTROL DIRECTIVE

The form of the single integer control directive, INT24, is:

```
INT24 v [,v] ...
```

where INT24 specifies a 24-bit integer data type and
v is the symbolic name of a variable or array.

Extract from CRAY-1 Fortran Reference Manual, 1978

47

Top Programming Languages

1. Java
2. C
3. C++
4. Python
5. C#
6. Javascript
7. PHP
8. Ruby
9. R
10. Matlab
11. SQL
12. Assembly
13. PERL
14. Visual Basic
15. HTML
16. Objective C
17. Shell
18. Arduino
19. Scala
20. Go
21. Processing
22. D
23. Lua
24. **Fortran**
25. Haskell
26. Lisp
27. VHDL
28. Delphi
29. Prolog
30. SAS
31. Clojure
32. ASP.Net
33. Verilog
34. Erlang
35. Cobol
36. Ada
37. Scheme
38. CoffeeScript
39. TCL
40. Actionscript
41. ABAP
42. Ladder logic
43. Ocaml
44. Apex Code
45. J
46. Eiffel
47. Forth
48. Scilab
49. NetLogo

IEEE Spectrum's 2014 Ranking

48

Languages / APIs Analysis Grid

- Base language(s)
- Main constructs (syntax and model)
- Underlying assumptions
- Some issues not covered
- Interactions with other APIs (e.g. MPI)
- Backtracking cost (e.g. cost of rewriting)
- Life expectancy, maintenance cost, portability (syntax and performance)
- Ecosystem (e.g. libraries, debuggers)
- Who's driving the changes

49

Cuda (2)

- Underlying assumptions
  - Can afford data copying (i.e. data can reside on the accelerator)
  - Loop type parallelism, plenty of fine grain parallelism available
  - Uniform threads body behavior (i.e. little control flow divergence)
- Some issues not covered
  - Partial handling of multiple devices (not in the // model)
  - MIMD parallelism (but multiple kernels possible)
  - Non C++ languages (except for Cuda Fortran from PGI)
  - Data structures deep copies

50

Cuda (3)

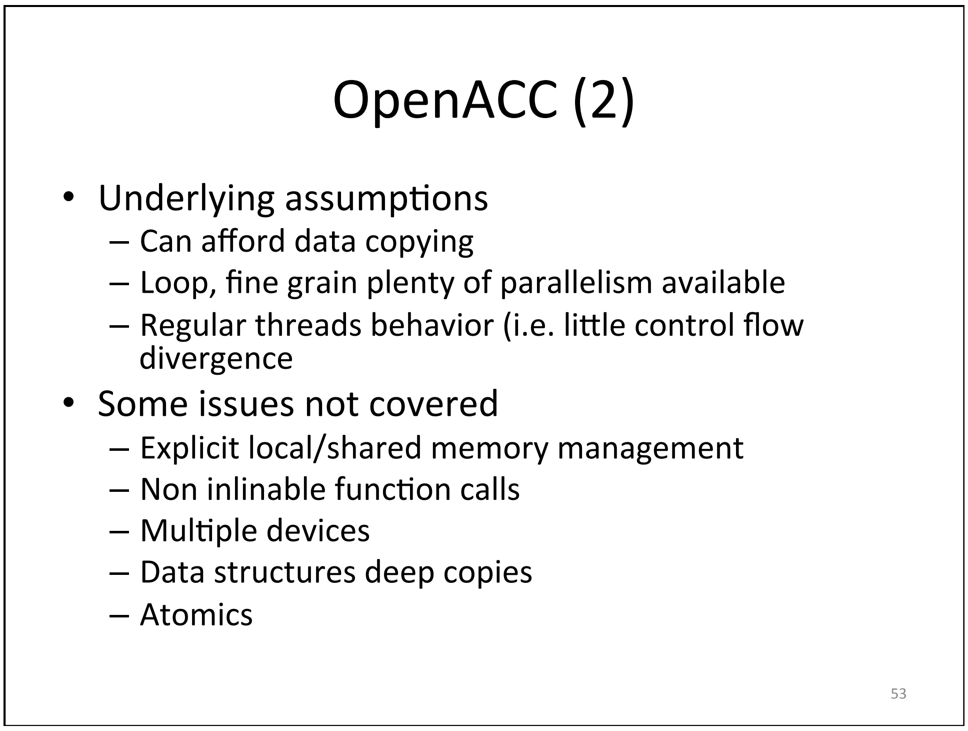
- Backtracking cost
  - Back to OpenMP not (very) costly
- Life expectancy, maintenance cost, portability
  - Not expected to disappear soon
    - Great for teaching
  - Not portable, re-tuning specific to device
  - Maintenance higher than regular codes (e.g. data management)
- Ecosystem
  - Rich libraries, performance tools
- Who's driving the changes
  - Nvidia

51

OpenCL (2)

- Backtracking cost
  - Back to OpenMP not (very) costly but higher than for Cuda (more runtime)
- Life expectancy, maintenance cost, portability
  - Not expected to disappear soon
  - Portable syntax (about performance later)
  - Available on most parallel devices (e.g. CPU, GPUs, FPGA)
  - Similar maintenance cost as Cuda
- Ecosystem (e.g. libraries, debuggers)
  - Limited especially debuggers
- Who's driving the changes
  - Khronos association, not HPC people

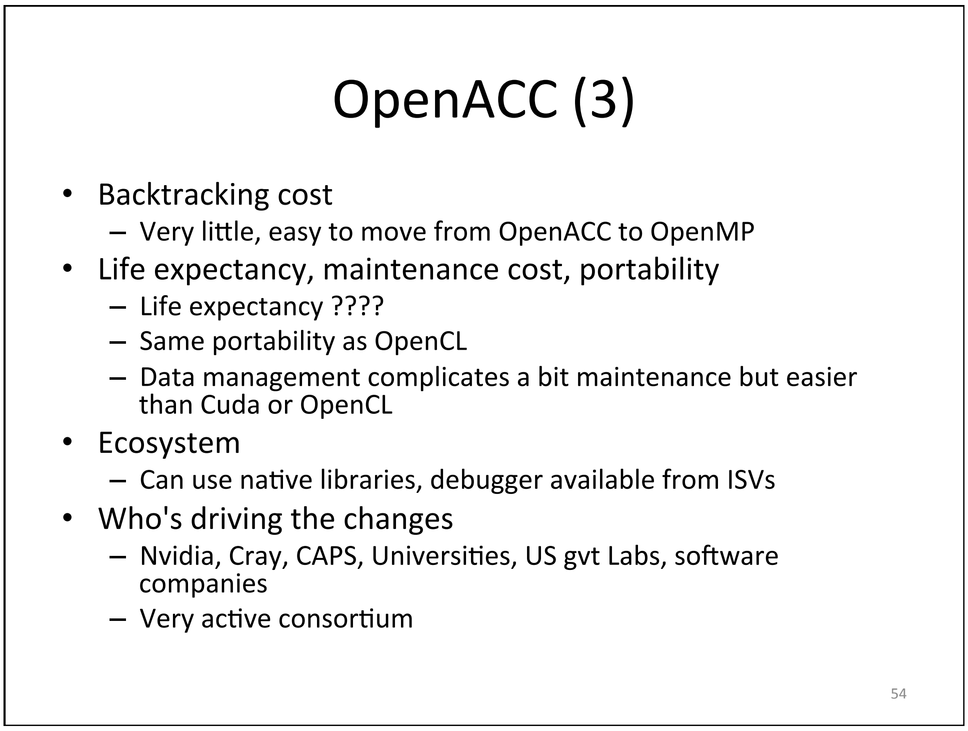
52



OpenACC (2)

- Underlying assumptions
  - Can afford data copying
  - Loop, fine grain plenty of parallelism available
  - Regular threads behavior (i.e. little control flow divergence
- Some issues not covered
  - Explicit local/shared memory management
  - Non inlinable function calls
  - Multiple devices
  - Data structures deep copies
  - Atomics

53



OpenACC (3)

- Backtracking cost
  - Very little, easy to move from OpenACC to OpenMP
- Life expectancy, maintenance cost, portability
  - Life expectancy ????
  - Same portability as OpenCL
  - Data management complicates a bit maintenance but easier than Cuda or OpenCL
- Ecosystem
  - Can use native libraries, debugger available from ISVs
- Who's driving the changes
  - Nvidia, Cray, CAPS, Universities, US gvt Labs, software companies
  - Very active consortium

54

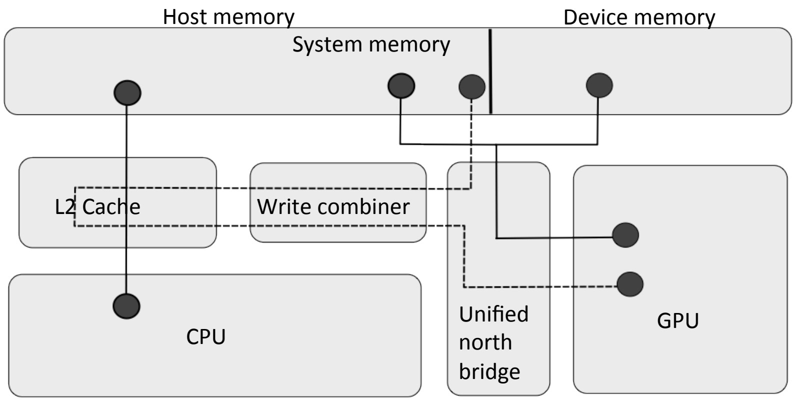
Synthesis (2)

- Libraries non uniform from one platform to another one
  - As important as the API / lang. for many applications
- Offload model threaten by small-big core multicores (or individual core DVFS)
- OpenACC and OpenMP accelerator potential convergence?
- HSA adoption broadness?
  - Heterogeneous System Architecture
  - Founders: AMD, ARM, Qualcomm, TI, …
  - Building a heterogeneous compute software ecosystem
  - http://hsafoundation.com
- Ecosystem still poor, need more ISVs, more visibility, more time

55

AMD Fusion APUs

- Accelerated Processing Architecture
- Removes PCI-X bottleneck



----: coherent memory path —— : non coherent memory path

56