Parallel Programming Languages and Approaches

Dr Alan D Simpson
Technical Director, EPCC
a.simpson@epcc.ed.ac.uk
+44 131 650 5120

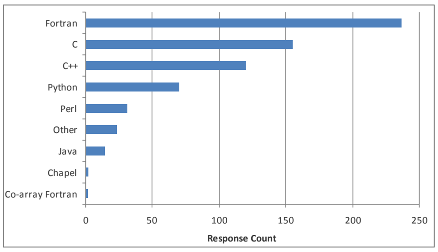
Contents

Contents

- A Little Bit of History
  - Non-Parallel Programming Languages
  - Vector Processing
  - Data Parallel
  - Early Parallel Languages
- Current Status of Parallel Programming
  - Parallelisation Strategies
  - Mainstream HPC
- Alternative Parallel Programming Languages
  - Single-Sided Communication
  - PGAS
  - Accelerators
  - Hybrid Approaches
- Final Remarks and Summary

Non-Parallel Programming Languages

- Serial languages are also important for HPC
  - Used for much scientific computing
  - Basis for parallel languages
- PRACE Survey results:



- PRACE Survey indicates that nearly all applications are written in:
  - Fortran: well suited for scientific computing
  - C/C++: allows good access to hardware
- Supplemented by
  - Scripts using Python, PERL and BASH
  - PGAS languages starting to be used

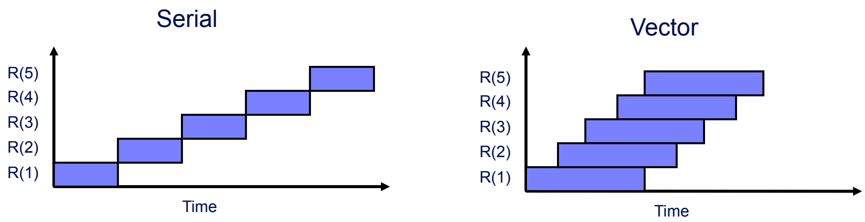
Vector Programming

- *Exploit hardware support for pipelining*
  - *and for fast data access*
- Early supercomputers were often vector systems
  - Such as the Cray-1
- Allowed operations on vectors
  - A vector is a a series of values
  - e.g., a section of a Fortran array
- Typical vector loop

```
DO i = 1, n
   y(i) = a*x(i) + y(i)
END DO
```

Vector Multiply

- Multiply process is made up of a number of stages
- Vector hardware allows stages to work independently and to pass results to each other in an “assembly line” manner
- Start-up cost as pipeline fills, but then a result every cycle



Vectorisation

- Sometimes required restructuring of loops to allow efficient vectorisation
- Directives used to provide information to the compiler about whether a particular operation was vectorisable
- Compilers became increasingly good at spotting opportunities for vectorisation
- Vector supercomputers became less popular as parallel computing grew
- However, many modern CPUs contain vector-like features
  - e.g., INTEL IvyBridge processors in Cray XC30

Data Parallel

- *Processors perform similar operations across data elements in an array*
- Higher level programming paradigm, characterised by:
  - single-threaded control
  - global name space
  - loosely synchronous processes
  - parallelism implied by operations applied to data
  - compiler directives
- Data parallel languages: generally serial language (e.g., Fortran 90) plus
  - compiler directives (e.g., for data distribution)
  - first class language constructs to support parallelism
  - new intrinsics and library functions
- Paradigm well suited to a number of early (SIMD) parallel computers
  - Connection Machine, DAP, MasPar,…

Data Parallel II

- Many data parallel languages implemented:
  - Fortran-Plus, DAP Fortran, MP Fortran, CM Fortran, *LISP, C*, CRAFT, Fortran D, Vienna Fortran
- Languages expressed data parallel operations differently
- Machine-specific languages meant poor portability
- Needed a portable standard: High Performance Fortran
- Easy to port codes to, but performance could rarely match that from message passing codes
  - Struggled to gain broad popularity

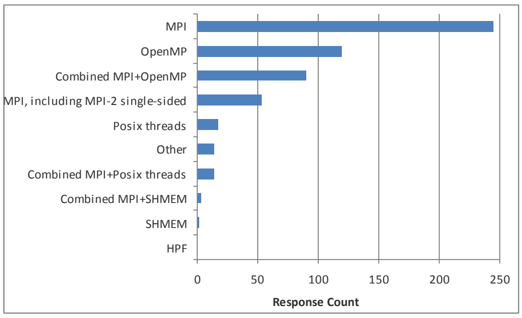
Early Parallel Languages

- Connection Machine languages
  - Thinking Machines Corporation provided *data parallel* versions of a variety of sequential languages (*LISP, C*, CM-Fortran)
  - Allowed users to exploit a large number of simple processors in parallel
- OCCAM
  - Early message passing language
    - …based on Communicating Sequential Processes
  - Developed by INMOS for programming Transputers
  - Explicitly parallel loops via PAR keyword
  - Language constructs for sending and receiving data through named channel
    - Could only communicate with neighbouring processors
    - → Message routing had to be in user software
- Most early languages for parallel computing were vendor-specific

Contents

Parallelisation Strategies

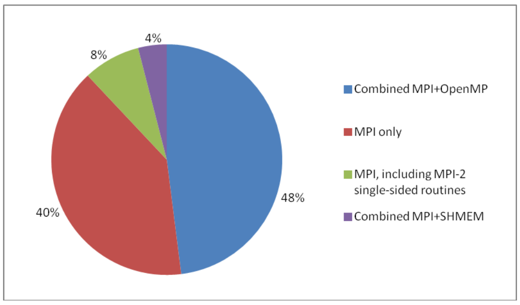
- PRACE asked more than 400 European HPC users
  - *"Which parallelisation implementations do you use?"*



  - Unsurprisingly, most popular answers were MPI, OpenMP and Combined MPI+OpenMP
    - Some users of Single-Sided communications

Parallelisation Strategies II

- PRACE also asked users of very largest systems:
  - *"Which parallelisation method does your application use?"*



- Most popular: "MPI Only" and "Combined MPI+OpenMP"
- 12% used single-sided routines

Mainstream HPC

- For the last 15+years, most HPC cycles on large systems have been used to run MPI programs, written in Fortran or C/C++
  - Plus OpenMP used on shared memory systems/nodes
- MSc in HPC includes compulsory courses in MPI and OpenMP
- However, there are now reasons why this may be changing:
  - Currently, HPC systems have increasingly large numbers of cores, but the individual core performance is relatively static
  - There are new challenges in exploiting future Exascale systems
- So, alongside mainstream HPC, there is also significant activity in:
  - Single-sided communication
  - PGAS languages
  - Accelerators
  - Hybrid approaches
- *Many of these areas are discussed later in this course*

Shared Memory

- *Multiple threads sharing global memory*
- Developed for systems with shared memory (MIMD-SM)
- Program loop iterations can be distributed to threads
  - Each thread can refer to private objects within a parallel context
- Implementation
  - Threads map to user threads running on one shared memory node
  - Extensions to distributed memory not so successful
- Posix Threads/PThreads is a portable standard for threading
- Vendors had various shared-memory directives
- OpenMP developed as common standard for HPC
  - OpenMP is a good model to use within a node
  - More recent task features

Message Passing

- *Processes cooperate to solve problem by exchanging data*
- Can be used on most architectures
  - Especially suited for distributed memory systems (MIMD-DM)
- The message passing model is based on the notion of *processes*
  - *Process*: an instance of a running program, together with the program's data
- Each process has access only to its own data
  - i.e., all variables are private
- Processes communicate with each other by sending+receiving messages
  - Typically library calls from a conventional sequential language
- During the 1980s, there was an explosion in message passing languages and libraries
  - CS Tools, OCCAM, CHIMP (developed by EPCC), PVM, PARMACS, …

MPI: Message Passing Interface

- *De facto* standard developed by working group of around 60 vendors and researchers from 40 organisations in USA and Europe
  - Took two years
  - MPI-1 released in 1993
  - Built on experiences from previous message passing libraries
- MPI's prime goals are:
  - To provide source-code portability
  - To allow efficient implementation
- MPI-2 was released in 1996
  - New features: parallel I/O, dynamic process management and remote memory operations (single-sided communication)
- Now, MPI is used by nearly all message-passing programs

Contents

- A Little Bit of History
  - Non-Parallel Programming Languages
  - Vector Processing
  - Data Parallel
  - Early Parallel Languages
- Current Status of Parallel Programming
  - Parallelisation Strategies
  - Mainstream HPC
- Alternative Parallel Programming Languages
  - Single-Sided Communication
  - PGAS
  - Accelerators
  - Hybrid Approaches
- Final Remarks and Summary

Single-Sided Communication

- *Allows direct access to memory of other processors*
  - Each process can access total memory, even on distributed memory systems
- Simpler protocol can bring performance benefits
  - But requires thinking about synchronisation, remote addresses, caching...
- Key routines
  - PUT is a remote write
  - GET is a remote read
- Libraries give PGAS functionality
- Vendor-specific libraries
  - SHMEM (Cray/SGI), LAPI (IBM)
- Portable implementations
  - MPI-2, OpenSHMEM

Single-Sided Communication

- Single-sided communication is major part of MPI-2 standard
  - Quite general and portable to most platforms
  - However, portability and robustness can have an impact on latency
  - Quite complicated and messy to use
- Better performance from lower-level interfaces, like SHMEM
  - Originally developed by Cray but a variety of similar implementations were developed on other platforms
  - Simple interface but hard to program correctly
- OpenSHMEM
  - New initiative to provide standard interface
  - See **`http://www.openshmem.org`**

PGAS: Partitioned Global Address Space

- *Access to local memory via standard program mechanisms plus access to remote memory directly supported by language*
- The combination of access to all data plus also exploiting locality could give good performance and scaling
- Well suited to modern MIMD systems with multicore (shared memory) nodes
- Newly popular approach initially driven by US funding
  - Productive, Easy-to-use, Reliable Computing System (PERCS) project funded by DARPA's High Productivity Computing Systems (HPCS)

PGAS II

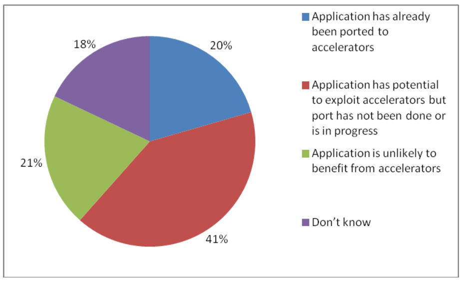
- Currently active and enthusiastic community
- Very wide variety of languages under the PGAS banner
  - See `http://www.pgas.org`
  - Including: CAF, UPC, Titanium, Fortress, X10, CAF 2.0, Chapel, Global Arrays, HPF?, …
- Often, these languages have more differences than similarities…

PGAS Languages

- The broad range of PGAS languages makes it difficult to choose which to use
- Currently, CAF and UPC are probably most relevant as Cray's compilers and hardware now support CAF and UPC in quite an efficient manner
- CAF: Fortran with Coarrays
  - Minimal addition to Fortran to support parallelism
  - Incorporated in Fortran 2008 standard!
- UPC: Unified Parallel C
  - Adding parallel features to C

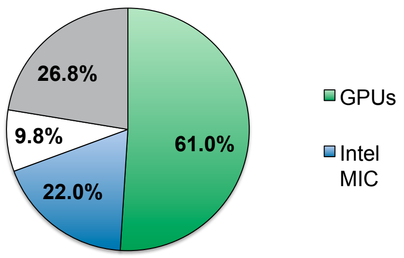
Accelerators

- *Use accelerator hardware for faster node performance*
- Recently, most HPC systems are increasing the number of cores, but individual cores are not getting much faster
  - This gives significant scaling challenges
- Accelerators are increasingly interesting
  - …for some applications
- PRACE Survey
  - *"Could your application benefit from accelerators, such as GPGPUs?"*
  - 61% thought so

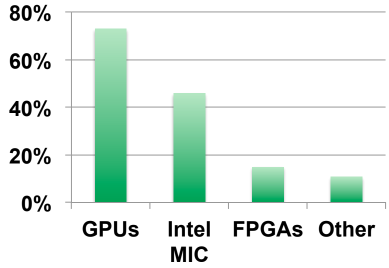


UK Usage of Accelerators

Current Usage



Future Plans



- FPGAs were very fashionable a few years ago
  - But proved difficult to program
- Currently, most interest is around GPUs
  - …with Intel MIC/Xeon Phi increasing in popularity too
- Most users think accelerators will continue to become more important

Programming GPUs

- Graphics Processing Units (GPUs) have been increasing in performance much quicker than standard processor cores
- Led to an interest in GPGPU (General Purpose computation on Graphics Processing Units)
  - Where the GPU acts as an *accelerator* to the CPU
- Variety of different ways to program GPGPUs
  - CUDA
    - NVIDIA's proprietary interface to the architecture.
    - Extensions to C (and Fortran) language which allow interfacing to the hardware
  - OpenCL
    - Cross platform API
    - Similar in design to CUDA, but lower level and not so mature
  - OpenACC
    - Directives-based approach
    - OpenMP-style directives - abstract complexities away from programmer

Hybrid Approaches

- *Use more than one parallelisation strategy within a single program*
- Trying to obtain more parallelisation by exploiting hierarchical parallelisation
- Most commonly, combining MPI + OpenMP
  - Using OpenMP across a shared-memory partition, with MPI to communicate between partitions
  - May make sense to use OpenMP just within a many-core processor
- But can also combine MPI with Pthreads or OpenSHMEM or CAF…
- Using many GPGPUs also often requires the use of MPI alongside the GPGPU programming approach

Contents

Why do Languages Survive or Die?

- It is not always entirely clear why some languages and approaches thrive while others fade away…
- However, languages which survive do have a number of common characteristics
  - Appropriate model for current hardware
  - Good portability
  - Ease of use
  - Applicable to a broad range of problems
  - Strong engagement from both vendors and user communities
  - Efficient implementations available

Summary

- Development of portable standards have been essential for uptake of new parallel programming ideas
- Mainstream HPC is currently based on MPI and OpenMP
  - However, there are alternatives
- Exascale challenges have injected new life into development of novel parallel programming languages and approaches
- The remainder of this course focuses on PGAS languages and programming GPGPUs
  - Plus lectures on data parallel programming and single-sided communication

References

- PRACE-PP
  - **D6.1: Identification and Categorisation of Applications and Initial Benchmarks Suite,** *Alan Simpson, Mark Bull and Jon Hill, EPCC*
- PRACE-1IP
  - **D7.4.1: Applications and user requirements for Tier-0 systems,** *Mark Bull (EPCC), Xu Guo (EPCC), Ioannis Liabotis (GRNET)*
  - **D7.4.3: Tier-0 Applications and Systems Usage,** *Xu Guo, Mark Bull (EPCC)*
- ARCHER User Requirements
  - Project Working Group: *Katharine Bowes (EPSRC), Ian Reid (NAG), Simon McIntosh-Smith (Bristol), Bryan Lawrence (NCAS/Reading) and Alan Simpson (EPCC)*