Parallel Programming Strategies

Eng. Fotis Georgatos <fotis@cyi.ac.cy>

Cyprus Institute, CaSToRC

joint PRACE/LinkSCEEM-2 CyI 2011 Winter School
24 January 2011
Fog of terms

- Serial and Parallel Tasks, Serial and Parallel Execution
- Symmetric Multi-Processor (SMP), Multi-core Processors
- Observed Speedup, Parallel Overhead, Scalability
- Shared Memory, Distributed Memory, Pipelining
- Granularity, Fine-grained, Coarse-grained
- Communications, Synchronization, Embarrassingly Parallel
- Massively Parallel, Cluster Computing, Supercomputing
TOP500.ORG: Performance in FLOPS, November 2010

- Moore’s law alive for 4 decades
TOP500.ORG: Number of Processors, November 2010

Note:
- Increasing # cores
- Smallest system now @ 2048 cores
- You MUST parallelize.
BLAS: Basic Linear Algebra Subprograms

BLAS: Linear Algebra library for vector/matrix operations:

- **BLAS-1**: Vector-Vector (SAXPY, DAXPY, DDOT et al)
- **BLAS-2**: Vector-Matrix (C-, Z-, S-, DGEMV et al)
- **BLAS-3**: Matrix-Matrix (SGEMM, DGEMM et al)

As an example, look at **DGEMM** - used in LINPACK/TOP500:

- D stands for Double-Precision Floating Point
- GE stands for General Matrix (not sparse or special)
- MM stands for Matrix Multiply (among many others)

If you can write better code than in BLAS, let us know.
If you don’t, prefer to reuse BLAS from within your code.
Parallel Programming Strategies

Take away messages ;-)

▶ Utilize existing HPC numerical kernels; Examples:
  ▶ Linear algebra: BLAS & BLACS, LIN-/LAPACK, ScaLAPACK
  ▶ Spectral methods: FFTW with parallel extensions
  ▶ Libraries: Trilinos, MKL (supersets of previous libraries)
  ▶ GNU Scientific Library
  ▶ Graph algorithms: CILK

▶ Utilize existing HPC commercial or community codes
  ▶ BQCD, NAMD, CPMD, CP2K, WRF, NEMO, NS3D, GPAW
  ▶ ECHAM5, BSIT, ELMER, SIESTA, GAUSSIAN, AMBER
  ▶ GAMESS, LAMMPS, CHROMA, FLUENT, SATURNE
  ▶ COSMO, VASP, Chipster, Parallel BLAST etc
  ▶ Octave with parallel extensions, R with GPU extensions etc

▶ Write one. See rest of this presentation and Winter School

So: if you have choice DO NOT re-write parallel code
Software Quality aspects

► Respect/be aware of standards
  ► Programming: ANSI C, ISO C90/99, FORTRAN ISO 90 etc
  ► Numerical: IEEE-754, IEEE 754-2008
  ► System: POSIX compliance

► Respect/be aware of scientific data formats
  ► HDF5 & BioHDF (this can help in Visualization, too)
  ► NetCDF
  ► GRIB, FITS, CERNLIB, XMDF et al

► Do Fault Tolerance and Verification & Validation (V&V)
► Do checkpointing; save the intermediate application states
► Documentation

YOU ARE THE USER OF THE SOFTWARE
AND YOU MUST ENSURE ITS QUALITY
Order is from "lowest hanging fruit" option, up to most expensive:

1. trivial parallelism ("embarrassing" eg. in parameter sweeps)
2. community software (code reuse)
3. community libraries (code reuse; SWIG may be relevant here)
4. custom code w. OpenMP (SMP exploitation)
5. custom code w. MPI (perhaps w. MPI I/O)
6. custom code w. Hybrid (MPI & OpenMP)
7. custom code in a PGAS language
8. Accelerators: GPUs; languages: CUDA & OpenCL
9. Accelerators: FPGAs; languages: low-level/custom
10. Accelerators: ASICs; languages: low-level/custom

NOTA BENE: You can take advantage of first 3 cases, with plain work in a scripting language.
Your adversary: HPC Fabric circuits

An integer section of an ALU, Arithmetic Logical Unit

- Either in an CPU, FPU or Network Interconnects, your application **will** face a bottleneck, somewhere.
- Constrained by speed of light and VLSI design limits
Abstraction layers in modern systems

You will need to tame the following layers:

- **Software**
  - Application
  - Algorithms/Libraries
  - Programming Language
  - Compilers/Interpreters - Theoretically you stop here
  - Operating System/Virtual Machines

- **Hardware**
  - Instruction Set Architecture - This may get relevant
  - Micro-architecture
  - Gates/Register-Transfer Level
  - Circuits
  - Devices
  - Physics

Understand Software/Computer Engineering & Computer Science
Flynn’s taxonomy, with diagrams

- **SISD**: Z-80 architecture
  - Single Instruction
  - Single Data
- **SIMD**: GPU cards
  - Single Instruction
  - Multiple Data
- **MISD**: Tandem NonStop
  - Multiple Instruction
  - Single Data
- **MIMD**: Modern HPC nodes
  - Multiple Instruction
  - Multiple Data
Memory organization taxonomy

- Shared memory
  - UMA = Uniform Memory Access:
    - memory address position DOES NOT affect access time
  - NUMA = Non-UMA:
    - memory address position DOES affect access time
  - cc-NUMA = cache-coherent NUMA:
    - common case with modern SMP systems
- Distributed memory: nodes have distinct memory spaces
- Hybrid, where Shared is combined with Distributed
NUMA: Non Uniform Memory Access

Hybrid with two Uniform Memory Access systems

- Modern Symmetric Multiprocessing Systems are ccNUMA
Performance evaluation

- Best serial algorithm: $T_s$
- Parallel algorithm: $T_p$
- **Speedup**: $S = T_s/T_p$
- How much faster is parallel code from its serial counterpart?
  - perfect linear speedup $\Rightarrow S = p$ (for $p$ processors)
  - sublinear speedup $\Rightarrow S < p$ (common case)
  - superlinear speedup $\Rightarrow S > p$ (requires investigation)
- **Efficiency**: $E = S/p$
  - Measures how successful parallelization work has been
  - Typically $E \leq 1$
Amdahl’s law

- **Definitions:**
  - Best serial algorithm: $T_s$
  - Parallel algorithm: $T_p$
  - $f$ is the fraction of serial algorithm that CANNOT be parallelized

- **Amdahl’s law:** $T_p = fT_s + (1-f)rac{T_p}{p}$
  - Consequently:
    Only optimize the fraction of code which can run in parallel
Optimization mandate

"premature optimization is the root of all evil..."
as stated by Donald Knuth

only optimize after having tracked a convincing case of
code bottleneck

focus on pieces of code which will unavoidably keep your
ALUs/chips busy and are worthy of attention

optimizations may turn redundant by seemingly irrelevant
changes in other parts of the code

normally, such work should only take place within
computational kernels
HPC applications often share common algorithmic sections

**Seven Dwarves** by P. Collela is a very common classification:

1. Dense Linear Algebra (eg. Matrix multiplications)
2. Sparse Linear Algebra (unlike Dense LA, zeros dominate)
4. N-Body Methods (eg. force calculation algorithms)
5. Structured Grids (eg. Lattice Boltzmann methods for CFD)
6. Unstructured Grids (eg. Finite Element Analysis)
7. MapReduce (incl. Monte Carlo methods)

Certainly the previous list can be amended; examples:

- Combinational Logic (eg. brute force cryptography)
- Graph Traversal (incl. sorting techniques, compilation)
- Dynamic Programming (eg. matching strings algorithms)
- Backtrack & Branch-and-Bound (eg. pruning TSP solutions)
- Graphical Models, Finite State Machines etc
Why parallel programming is hard? overview

You may need to take care/ensure of all of the following:

▶ Respect memory hierarchy
▶ Tune for sequential performance
▶ Tune for parallel performance

Put it all together:

▶ and still have adequate numerical stability
▶ and still have a maintainable code.
Why parallel programming is hard? in detail

You may need to take care/ensure any of the following:

- ACCESS COLUMNs vs ROWs
  - access data in order of storage
- PREFETCH
  - feed L1, L2, L3 caches with data at just the right pace
- LOOP UNROLLING
  - prefetch instructions for optimal pipelining
- BENCHMARKING
  - understand performance curves of your code
- PARTITIONING:
  - DOMAIN/FUNCTIONAL DECOMPOSITION
  - break up the problem at just the right pieces
- PARALLEL COMMUNICATION
  - program using communication primitives (MPI, OpenMP)
- RESOURCE BALANCING
  - communications/computations should overlap

In which order to optimize? What on Visualization? Debugging?
What is memory hierarchy?

Think of the following as pyramid of facilities, of increasing space:

- Registers, $O(\text{bytes})$
- L1 cache, $O(\text{Kbytes})$
- L2 cache, $O(\text{Mbytes})$
- L3 cache, $O(10\text{Mbytes})$
- RAM Random Access Memory, $O(\text{GBytes})$
- SWAP (w. RAM are addressed jointly in Virtual Memory)
- Hard Disk Storage, $O(\text{TBytes})$
- Tape Storage, $O(100 \text{TBytes})$

We need to ensure that data-flows match the hierarchy
Memory Hierarchy basics

- Memory hierarchy exists because
  - fast memory is expensive and has to be small
  - slow memory is cheap and can be big

- Latency
  - How long do I have to wait for the data
  - Measured in micro/nanoseconds

- Throughput
  - What is the rate of incoming data?
  - Measured in GBs or MBs/second

- Total time = latency + (amount of data / throughput)
### Memory Hierarchy latencies

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Latency (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>Compress 1K bytes with Zippy</td>
<td>3,000</td>
</tr>
<tr>
<td>7</td>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>8</td>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>9</td>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>10</td>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>11</td>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
<tr>
<td>12</td>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>
**COLUMNS vs ROWS**

**Loop normal**

```
DO I = 1, N
   DO J = 1, M
      A(I, J) = B(I, J) + C(I, J)
   ENDDO
ENDDO
```

**Loops interchanged**

```
DO J = 1, M
   DO I = 1, N
      A(I, J) = B(I, J) + C(I, J)
   ENDDO
ENDDO
```
INTEL’s website reports about 'PREFETCHNTA' x86 command:\footnote{http://software.intel.com/en-us/articles/use-software-data-prefetch-on-32-bit-intel-architecture/}

- Use software data prefetch to hide the latency of data access in performance-critical sections of application code.
- The prefetch instruction allows data to be fetched in advance of its actual usage.
- The prefetch instructions do not change the user-visible semantics of a program, although they may affect the program's performance.
- The prefetch instructions merely provide a hint to the hardware and generally will not generate exceptions or faults.
Parallel Programming Strategies

Development of parallel code

LOOP UNROLLING

LOOP UNROLLING

Loop normal

1    DO  I=1,N
2       A(I) = A(I) + B(I) * C
3    ENDDO

Loop unrolled

1    DO  I=1,N,4
2       A(I) = A(I) + B(I) * C
3       A(I+1) = A(I+1) + B(I+1) * C
4       A(I+2) = A(I+2) + B(I+2) * C
5       A(I+3) = A(I+3) + B(I+3) * C
6    ENDDO
A climate model is natural to be split into its components; Domain & Functional Decomposition could be combined.
An interesting artifact of an LQCD code from yr2004

What kind of computer system would you buy and why?

2http://www.usqcd.org/fnal/talks/FNALcluster.pdf
Matrix Multiply can be decomposed to self-similar operations

- Smaller tiles are themselves Matrices of same type
Practicalities to consider: Part I

Do you need communications?

- No: embarrassingly parallel (fi. correlation of datasets)
- Yes: Common in HPC activities (fi. 3-D heat diffusion)

Factors to consider:

- cost of communications; subsystem overheads
- latency vs bandwidth
  - latency; 100 nanoseconds for QDR Infiniband fabrics
  - bandwidth; 40Gbps p2p for QDR Infiniband fabrics
Who controls communications?
  ▶ Message Passing Model (Explicit, programmer dictates them)
  ▶ Data Parallel Model (Implicit, driven by data demands)

Need to synchronize?
  ▶ Synchronous (blocking; imply handshaking messages)
  ▶ Asynchronous (non-blocking; allows cpu/io interleaving)

Scope of communications
  ▶ Point-to-Point
  ▶ Collective (broadcast, scatter, gather, reductions)
Synchronization types

- **Barrier**
  - tasks block when they reach the barrier
  - tasks synchronize when last one reaches the barrier

- **Lock/semaphore**
  - Like in OSs; used as primitive for atomic data/code access

- **Synchronous communication operations**
  - Used for complex cooperative communication patterns
Communication Primitives

- MPI; often combined with *coarse-grained* parallelism
- OpenMP; often combined with *fine-grained* parallelism
- Posix Threads (applicable well in full node allocations)
- TBB (C++ template library developed by Intel)
- SHMEM (shared memory access library - almost)
- GA → GAS → PGAS Languages

(slides coming soon)
MPI stands for Message Passing Interface

some important facts follow;

▶ an API specification that allows processes to communicate
▶ sending and receiving messages, put/get operations
▶ point-to-point and collective communication
▶ typically used for parallel programs
▶ binding for many languages, incl. C/C++ & Fortran
▶ serves equally well computer clusters and supercomputers
  ▶ where the cost of accessing non-local memory is high.
▶ MPI is expected to survive the Petascale & Exascale eras.
▶ **De facto ubiquitous standard in HPC systems**

MPI will be treated extensively during this week;
OpenMP

Pros:
- Useful for shared memory parallelism
- Suitable for small parallel sections (fine grained)
  - but, not too fine: choice of granularity still yours
- Try to optimizing on feeding L1/L2 caches at right pace

Cons:
- Scalability up to the level of a node
- No collective reductions like in MPI
Serial code example

```c
for (steps=0; steps < T; steps++)
    for (i=1; i<X-1; i++)
        for (j=1; j<Y-1; j++)
                                    A[step][i-1][j] +
                                    A[step][i+1][j] +
                                    A[step][i][j-1] +
                                    A[step][i][j+1])
```

Serial code without OpenMP
Parallel code example

```c
#pragma omp parallel
#pragma omp single
for (steps=0; steps < T; steps++){
    for (i=1; i<X-1; i++)
        for (j=1; j<Y-1, j++)
            #pragma omp task
                                   A[step][i-1][j] + 
                                   A[step][i+1][j] + 
                                   A[step][i][j-1] + 
                                   A[step][i][j+1])
            #pragma omp taskwait
    // Is fine-grained parallelism, perhaps, too fine? //
```
Marc Snir’s rules of proper shared memory programming:

1. Ensure that any two conflicting accesses to shared memory are synchronized
   Program behavior is undefined otherwise
2. Use ordering synchronization whenever possible
   - Problem: all parallel languages require proper synchronization;
     - no system enforces this requirement.
     - Race detections tools are incomplete.
     - Solutions are in research stage
   - SHMEM is commercial, provided for SGI & Cray systems
     applies the programming model on fast interconnects.
PGAS Languages

- PGAS stands for Partitioned Global Address Space
  - They are designed around memory hierarchy aspects
- PGAS model concepts are the basis of:
  - Unified Parallel C
  - Co-array Fortran
  - Titanium
  - Fortress
  - Chapel
  - X10
- They tend to provide more abstract programming notation
communications and computations should ideally overlap

formerly known in non parallel systems as "Balance Triangle"

PRACE D6.2.1: ECHAM5 general circulation model Kiviat diagram FLOPs rate and communications bandwidth are dominant.
Performance optimization tools

You are not alone:

- Prof/Gprof
- PAPI
- Intel Trace Analyzer
- CrayPat, Apprentice2
- Vtune
- Vampir
- Tau
- Kojac
- IPM
- Allinea OPT & DDT
This is an inexhaustible topic; Slide intentionally left blank.
Challenges of parallel programming

- Identify opportunities for parallelism
  - manual (by developer)
  - automated (by compiler)

- Express parallelism
  - low-level (pthreads, MPI)
  - high-level (OpenMP, UPC, Chapel, PGAS languages)

- Project parallelism
  - scheduling, creation, termination
  - operating system, runtime system

- Synchronization
  - coarse grained
  - fine grained
  - deadlock free
  - composable
Parallel Programming Strategies

Rules of thumb and common advices

Use the write tool for each purpose:

- **Prototyping**: use scripting languages, for code wrapping, too
- **Portability**:
  use a language known to **compile across architectures**
- **Speed**:
  use a language known to **compile well across architectures**
- **Architecture**:
  watch out for communication patterns and intermediate files
- **Modularity**: strive for modules approach; where possible
  provide minimalistic but sufficient Command Line interfaces
- **Extensibility**: Use multiple and well defined layers (APIs)
- **Code reuse**: if there's a tool known to work for your case;
  don't rewrite one (eg. PBLAS, NetCDF, HDF5)
- **Standards**: IEEE754, ANSI C/FORTRAN, POSIX et al
Conclusions

- Exploit performance at:
  - core
  - chip
  - node
  - communication fabric
  - system level

- Unfortunately no silver bullet exists

- Certain strategies exist, often successful but, tough to guarantee optimal performance. Research.

- Iterative ’Trial and error’ aspects dominate
Reference material

1. Introduction to Parallel Computing by LLNL
   https://computing.llnl.gov/tutorials/parallel_comp/

2. Designing and Building Parallel Programs, by Ian Foster
   http://www.mcs.anl.gov/~itf/dbpp/

3. Introduction to Parallel Computing by Grama-Gupta-Karypis-Kumar
   http://www-users.cs.umn.edu/~karypis/parbook/

4. PRACE D6-4 Report on Approaches to Petascaling
Thank you