Performance of deal.II on a node

Bruno Turcksin

Texas A&M University, Dept. of Mathematics

Outline

Introduction

- Often supercomputers for the big national laboratories drive the market (example: BlueGene) but not always (example: Roadrunner, x86 with Cell accelerators).
- New generation of supercomputers in the next few years:
  - Los Alamos will receive Trinity this year.
  - Oak Ridge, Livermore, and Argonne (CORAL program) announced big contracts for a new generation of supercomputers (pre-exascale $\approx 100 petaflops$).

Los Alamos: Trinity

Trinity specifications (Cray):

- $\approx 40 petaflops$.
- 9500 nodes with Xeon processors (16 cores and 32 threads, AVX2) in 2015.
- 9500 nodes with Knights Landing Xeon Phi (60 cores and 240 threads?, AVX-512) in 2016.

Oak Ridge and Livermore: Summit and Sierra

Summit specifications (IBM):

- $150 - 300 petaflops$.
- $\approx$ 3400 nodes.
- POWER9 CPU (POWER8 has 12 cores and 96 threads) and Volta GPU connected through NVLink in 2018.
- Multiple CPUs and GPUs per node.

Argonne: Aurora

Aurora specifications (Intel):

- $180 - 450 petaflops$.
- more than 50,000 nodes
- Knights Hill Xeon Phi in 2018.

Architecture summary

- CPU: more cores and many threads per core.
- Xeon Phi: large number of cores/threads but cores are simpler and slower. 10 times more threads than CPU but each core 1/4 to 1/3 the performance of Xeon and the code needs to be vectorized.
- GPU: very large number of very simple cores.

Vectorization

Vector or SIMD (Single Instruction Multiple Data) instruction apply the same operation simultaneously to several pieces of data. Ex.:

```
for (i=0; i<N; ++i)
  a[i] = b[i] + c[i]
```

Vectorization:

```
for (i=0; i<N; i+=2)
  a[i:i+1] = b[i:i+1] + c[i:i+1]
```

Loop unrolling:

```
for (i=0; i<N; i+=2)
  a[i] = b[i] + c[i]
  a[i+1] = b[i+1] + c[i+1]
```

Vectorization

- Current CPUs (Haswell and Broadwell) have instructions that work on 256 bits, i.e., on four double variables.
- Xeon Phi have instructions that work on 512 bits, i.e., on eight double variables $\Rightarrow$ important to take advantage of vectorization but cannot be done efficiently by compiler $\Rightarrow$ needs to annotate the loop.

Vectorization

Vectorization can be achieved only if there is no "forward" dependencies in the loop:

- this loop cannot be vectorized:

```
for (i=1; i<size; ++i)
  a[i] = 2*a[i-1];
```

- this loop can be vectorized:

```
for (i=0; i<size-1; ++i)
  a[i] = 2*[a+1];
```

Vectorization

Why the first loop cannot be vectorized? The code that will be executed is equivalent to

```
b=a;
for (i=1; i<size-1; i+=2)
  a[i] = 2*b[i-1];
  a[i+1] = 2*b[i];
```

This will give a different result than:

```
for (i=1; i<size-1; i+=2)
  a[i] = 2*a[i-1];
  a[i+1] = 2*a[i];
```

Vectorization

For the second loop, we have:

```
b = a;
for (i=0; i<size-2; i+=2)
  a[i] = 2*b[i+1];
  a[i+1] = 2*b[i+2];
```

which is the same as:

```
for (i=0; i<size-2; i+=2)
  a[i] = 2*a[i+1];
  a[i+1] = 2*a[i+2];
```

Vectorization

This cannot be vectorized by the compiler:

```
void add(double* a, double* b, double* c, int size)
{
  for (i=0; i<size; ++i)
    a[i] = b[i] + c[i];
}
```

Risk of hidden dependencies (Ex: $a[:] = b[1:]$) $\Rightarrow$ need compiler directive.

Europe and Japan

- Europe: ARM processors and embedded GPU.
- Japan: SPARC64 XIfx, 32 cores + 2 assistant cores for OS and MPI, 256 bits SIMD

CPU, Xeon Phi, and GPU

Performance of difference architectures:

- **K80:** 2,910 Gflops with TDP of 300 watts
- **Xeon Phi 7120:** 1,208 Gflops with TDP of 300 watts
- **Xeon (Haswell) E5-2699v3:** 662 Gflops with TDP of 145 watts (AVX2 and FMA3)

Need to increase performance per watt $\Rightarrow$ advantage of GPU and Xeon Phi.

In the future, all the architectures will used more threads so what is the problem?

CPU

GOAL: make a single thread as fast as possible.

- Large and hierarchical cache memories to reduce latency
- Branch predictor
- Out-of-order execution
- High frequency
- ...

Cores are complicated $\Rightarrow$ take more room $\Rightarrow$ few cores.

GPU

GOAL: throughput oriented

- Single thread "does not matter"
- Hide memory latency through parallelism $\Rightarrow$ small cache (it does not matter if a thread stalls because there are so many threads)
- Programmer has to deal with the storage hierarchy himself
- SIMD replaced by SIMT (Single Instruction Multiple Thread = several cores have to execution the same instruction)
- Low frequency
- ...

Cores are simple $\Rightarrow$ take less room $\Rightarrow$ a lot of cores.

Xeon Phi

Between a CPU and GPU:

- Cores based on old Pentium designed $\Rightarrow$ cores smaller than a regular CPU $\Rightarrow$ more cores
- No shared cache between cores but hardware-based cache coherency
- No out-of-order execution or branch prediction
- Use of vectorization is very important
- Low frequency

Can run the same code as a CPU but performance will probably be bad.

Libraries

How to write code for these new architectures?

- Problem with CUDA: not portable and a lot of work (need to learn CUDA...).
- Problem with OpenCL: portable across architectures but need to write platform specific codes to get the most performance.
- Problem with OpenMP 4.0: portable across architectures but support for GPUs still very new.

$\Rightarrow$ use a library with different backends.

Libraries

Which one?

- Kokkos: developed at Sandia and part of Trilinos, heavily templated C++ library
- Paralution: stand-alone library
- ViennaCl: stand-alone library
- OCCA2, OmpS, RAJA, VexCL, etc.

Paralution

- Library that has several iterative solvers, multigrid, and preconditioners for CPU, Xeon Phi, and GPU through OpenMP, OpenCL, and CUDA.
- Works on Linux, Windows, and Mac OS.
- Started at the Uppsala University with open source license (GPLv3).
- Moved to a company and more and more of the code became proprietary: multi-nodes capabilities, new AMG, etc. $\Rightarrow$ this is a problem.

Results

- Comparison between standalone deal.II (multithreading), Trilinos (MPI), and Paralution (multithreading/GPU) on a slightly modified step-40 (Laplace equation) using CG+SSOR and CG+AMG.
- n dofs $\approx 1.7 \cdot 10^6$
- Node with two sockets of 10-core Intel Xeon (Ivy Bridge) and K20m GPGPU.

Results: CG+SSOR

N threads	1	4	8	12	16	20	GPU
N iter	2408	2408	2408	2408	2408	2408	2408
Time (s)	272	110	65.2	54.7	47.4	47.7	31.5

Table: Paralution - SSOR

N cores	1	4	8	12	16	20
N iter	2430	2903	2928	2897	2944	2937
Time (s)	221	201	105.9	75.9	57.1	47.8

Table: Trilinos - SSOR

Results: CG+SSOR

N cores	1	4	8	12	16	20
N iter	2436	2436	2436	2436	2436	2436
Time (s)	201	158.9	159.2	149.4	154.5	151.1

Table: deal.II - SSOR

Results: CG+SSOR

- Paralution with GPU is much faster than Trilinos.
- deal.II does not scale even on small number of processors (solver and preconditioner use multithreading only through generic vector operations).

Results: CG+AMG

N threads	1	4	8	12	16	20	GPU
N iter	6	6	6	6	6	6	6
Time (s)	228	81.8	62.6	62.3	50.8	49.5	32.5

Table: Paralution - AMG

N cores	1	4	8	12	16	20
N iter	39	40	42	39	40	40
Time (s)	7.98	2.36	1.56	1.101	0.893	0.865

Table: Trilinos - AMG

Results: CG+AMG

- Trilinos is much faster than Paralution with GPU.
- Paralution AMG is worse than SSOR!

ViennaCL

- ViennaCL is a library with capabilities similar to Paralution but limited to one node (no MPI).
- Header only library $\Rightarrow$ do not need to compile it.
- Open source (MIT license).
- Support in PETSc for ViennaCL.
- Support for Linux, Mac OS X, and Windows.

ViennaCL

- Convergence using CG without preconditioner: 7123 iterations in 52.5s
- Still have problems with AMG but I expect a speed up of 10 $\Rightarrow$ still slower than Trilinos.
- Benchmark using ViennaCL AMG shows that GPUs have a hard time to compete against CPUs: own AMG is the fastest on CPU and the slowest on Xeon Phi.

AMGCL

- AMGCL is a library for constructing an algebraic multigrid hierarchy.
- Can use OpenMP, OpenCL, and CUDA.
- Header only library $\Rightarrow$ do not need to compile it.
- Open source (MIT license).
- The code offers multiple backends including ViennaCL.

AMGCL

Codes, given by Denis Demidov, solve 3D Laplace equation using finite difference, n dofs $\approx 2.1 \cdot 10^6$, (AMGCL is run on a K40)

N cores	1	4	8	12	16	20	AMGCL
N iter	9	8	9	9	9	9	23
Time (s)	13.5	7.5	2.0	1.5	1.3	1.8	2.9
Solve (s)	5.7	3.0	0.89	0.68	0.63	0.87	0.72

Table: Laplace 3D

Tpetra

- Epetra is slowly being phased out and is replaced by Tpetra.
- Tpetra uses a Kokkos whose is focused on performance portability using POSIX Threads, OpenMP, and CUDA.
- Kokkos requires C++11
- Kokkos not always deterministic $\Rightarrow$ two different runs can lead to two different results.
- Deal.II should have support for Tpetra in the coming months.

Conclusions

- Need to work on future architectures without committing to one $\Rightarrow$ need to use libraries.
- For now, AMG on GPU does not seem to compete with state-of-the-art AMG on CPU (successive grids are built on the CPU).
- Fortunately not all applications require MG preconditioner. Do you?
- Need more tests for matrix-free.

Conclusions

Xeon Phi:

- Much younger technology $\Rightarrow$ need to wait to see how it will compete with CPU.
- three stages of porting your applications to Xeon Phi: it's horrible, it's great, it's meh.

INSTALLATION OF DEAL.II ON A CLUSTER

Installation of deal.II

How to install deal.II on Linux cluster (not Cray or BG):

- "by hand"
- using Linuxbrew (thanks Denis)
- using CANDI (thanks Uwe)

I have not tried Linuxbrew so I will not talk about it. Installing deal.II using CANDI:

```
./candi.sh deal.II/platforms/supported/linux_cluster.platform
```

CANDI

- Can build the latest stable version of deal.II or the development version.
- Configuration file similar to DEAL.II configuration.
- Download and install all the third-party libraries.
- Let you link with the libraries that you have installed yourself.
- Let you choose the number of processors to use.
- Only tested with GCC.
- **Support for MKL**.

Default setting are for desktop users not for cluster (ex. DEAL_II_COMPONENT_PARAMETER_GUI is ON).