Introduction to PETSc

Performance

Loïc Gouarin

Laboratoire de Mathématiques d'Orsay

May 13-15, 2013

Outline

PETSc offers a lot of well-tested routines but it’s not enough to have good performance.

Users have to pay attention how to put these routines together in their application code to have good performance.

It’s important to check serial performance before to try to have good scaling.

Some hints found in the documentation

- Configure your code with `--with-debugging=0`.
- Insert several (many) elements of a matrix or vector at once.
- Perform good preallocation for sparse matrices.
- Don't use too many `PetscMalloc`.
- Data structures should be reused whenever possible.
- Test different solvers to have good convergence rate for your application.

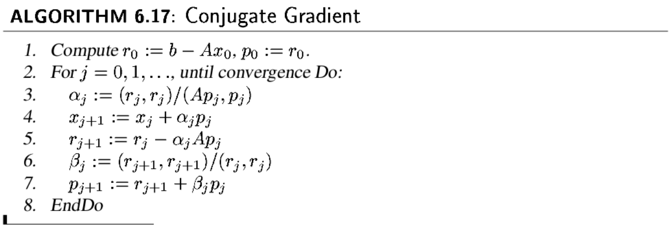
How to get information?

- `-log_summary`
  prints performance and memory consumption at program's conclusion.
- `-info [infofile]`
  prints verbose information about code.
- `-log_trace [logfile]`
  traces the beginning and ending of all PETSc events.
- `-malloc_log`
  keeps log of all memory allocations.

In the most cases, when you use linear and nonlinear solvers, you perform

- AXPY vector operations
- Matrix-vector products for sparse matrices
- Dot products

Conjugate gradient



ALGORITHM 6.17: Conjugate Gradient

```
1. Compute $r_0 := b - Ax_0$, $p_0 := r_0$.
2. For $j = 0, 1, \ldots$, until convergence Do:
3.     $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4.     $x_{j+1} := x_j + \alpha_j p_j$
5.     $r_{j+1} := r_j - \alpha_j Ap_j$
6.     $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7.     $p_{j+1} := r_{j+1} + \beta_j p_j$
8. EndDo
```

Figure: Iterative Methods for Sparse Linear Systems - Youcef Saad

AXPY performance

Let x and y two vectors of size n. We want to compute

$$y = y + \alpha x$$

with $\alpha = 1$.

AXPY performance

Let x and y two vectors of size n. We want to compute

$$y = y + \alpha x$$

with $\alpha = 1$.

- Computation

  1 ADD and 1 MULT $\Longrightarrow$ $2n$ flops

- Memory access

  2 READ and 1 WRITE $\Longrightarrow$ $3nb$ bytes

  where b is the size of the data (`float`: 4 bytes, `double`: 8 bytes, ...)

AXPY performance

Let x and y two vectors of size n. We want to compute

$$y = y + \alpha x$$

with $\alpha = 1$.

- Computation

  1 ADD and 1 MULT $\Longrightarrow$ $2n$ flops

- Memory access

  2 READ and 1 WRITE $\Longrightarrow$ $3nb$ bytes

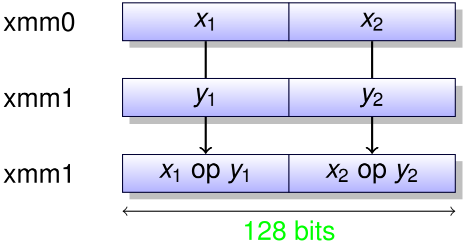
  where b is the size of the data (`float`: 4 bytes, `double`: 8 bytes, ...)

For $n \gg 1$, $3nb \gg 2n \Longrightarrow$ memory-bound problem

Theoretical peak FLOPS

On my laptop $\Longrightarrow$ Intel Core 2 (2.53GHz)

- 4 cores
- sse



$P_{peak} = 2.53 * 2 = 5.06 Gflops$ for one core.
$P_{peak_4} = 2.53 * 2 * 4 = 20.24 Gflops$ for 4 cores.

Memory bandwith

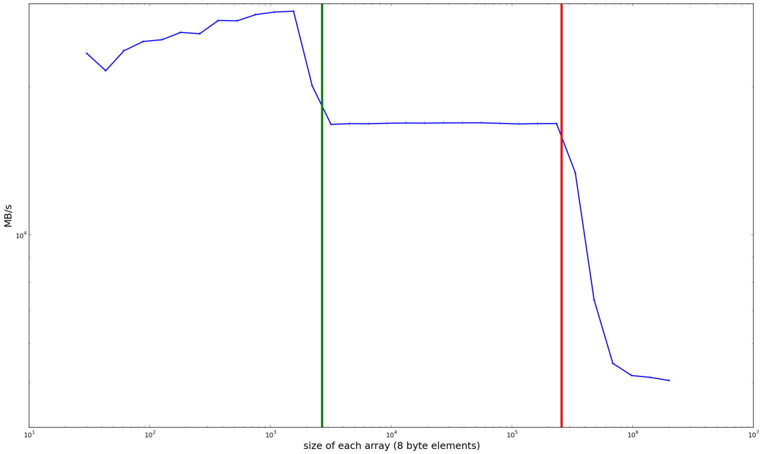
Stream2 benchmark

Kernel	Code	Bytes/ite read	Bytes/ite written	FLOPS/iter
FILL	$a(i) = q$	0 (+8)	8	0
COPY	$a(i) = b(i)$	8 (+8)	8	0
DAXPY	$a(i) = a(i) + qb(i)$	16	8	2
SUM	$sum = sum + a(i)$	8	0	1

http://www.cs.virginia.edu/stream/stream2/

Memory bandwith

DAXPY results



AXPY performance

For 2 operations, we need $3b$ memory access.
So to have P_{peak}, the memory bandwith required is

$$B_{req} = \frac{3bP_{peak}}{2} = 7.59b\ GB/s$$

Stream2 benchmark shows that the best memory bandwith on my laptop for sufficiently large values of n is

$$B_{peak} = 5.1 GB/s$$

So the maximum performance is

$$P_{max} = 5.1\frac{2}{3b} = 425\ Mflops$$

Exercise 1

- Extract `AXPY.tgz`.
- Go to the `AXPY/build` directory.
- Use cmake to configure, then make to build the project.
- Run `./stream2f`.
- Run `./AXPYtest -log_summary`.
- Check the results.

CSR format

- `n` number of lines
- `nnz` number of nonzero entries
- `row_ptr` array of integer
- `col_ind` array of integer
- `val` array of double

Example of CSR matrix

$$\begin{pmatrix} 1 & -2 & 0 & 0 & 0 \\ -4 & 1 & -2 & 0 & 0 \\ 0 & 2 & 5 & 0 & 2 \\ 0 & 0 & 1 & -3 & 3 \\ 8 & 0 & 0 & 2 & 1 \end{pmatrix}$$

We obtain:

$$\begin{aligned} n &= 5 \\ nnz &= 14 \\ row\_ptr &= \{0, 2, 5, 8, 11, 14\} \\ col\_ind &= \{0, 1, 0, 1, 2, 1, 2, 4, 2, 3, 4, 0, 3, 4\} \\ val &= \{1, -2, -4, 1, -2, 2, 5, 2, 1, -3, 3, 8, 2, 1\} \end{aligned}$$

Matrix-vector product

Perform $Ax = y$ with x an array of size n and A a square matrix with CSR format.

```
for (i=0; i<n; i++){
  sum = 0.;
  for(j=row_ptr(i);j<row_ptr(i+1);j++)
    sum += val[j]*x[col_ind[j]];
  y[i] = sum;
}
```

Matrix-vector product

Perform $Ax = y$ with x an array of size n and A a square matrix with CSR format.

```
for (i=0; i<n; i++){
  sum = 0.;
  for(j=row_ptr(i);j<row_ptr(i+1);j++) // 1 read
    sum += val[j]*x[col_ind[j]]; // 3 read, 2 flops
  y[i] = sum; // 1 write
}
```

Matrix-vector product

- Computation

$$2nnz\ flops$$

- Memory access

$$4n + 4nnz + 8nnz + 8n + 8n\ Bytes$$

The memory bandwith required is

$$B_{req} = \frac{10n + 6nnz}{nnz} P_{peak} GFlops$$

Recall that the best memory bandwith on my laptop for sufficiently large values of n is

$$B_{peak} = 5.1 GB/s$$

Then the maximum performance is

$$P_{max} = 5.1 \frac{nnz}{10n + 6nnz} Flops/s$$

For 2D Poisson problem with a five-point finite difference method, we have $\frac{n}{nnz} \approx \frac{1}{5}$, so

$$P_{max} = 637.5 MFlops/s$$

Exercise 2

- What is the memory used for the 2D Poisson problem using an assembled matrix ?
- Check the performance of your matrix-vector product for the 2D Poisson problem using an assembled matrix.
- Check the performance of your matrix-vector product for the 2D Poisson problem using a shell matrix.

The size of the matrix and the number of nonzero entries can be found with `-mat_view_info`.

How to define your own logs?

PETSc logs object creation, times, and floating-point counts for the library routines.
Users can easily add events to monitoring their application codes.

```
int USER EVENT;
PetscLogEventRegister("User event name",0,&USER EVENT);
PetscLogEventBegin(USER EVENT,0,0,0,0);
/* application code segment to monitor */
PetscLogFlops(number of flops for this code segment);
PetscLogEventEnd(USER EVENT,0,0,0,0);
```

Profiling multiple sections of code

By default, PETSc provides one profile listing. You can add logging stage

```
PetscLogStageRegister("Stage 1 of Code",&stagenum1);
PetscLogStagePush(stagenum1);
/* stage 1 of code here */
PetscLogStagePop();
PetscLogStageRegister("Stage 2 of Code",&stagenum2);
PetscLogStagePush(stagenum2);
/* stage 2 of code here */
PetscLogStagePop();
```

Exercise 3

- Add `PetscLogFlops` in your matrix-vector product using shell matrix and check the performance.
- Add multiple stages to identify each portion of you application: matrix creation, solver, ...

References

1. Matthew Knepley's tutorial (Serial Performance Part)
   http://www.mcs.anl.gov/petsc/documentation/tutorials/GUCASTutorial10
2. D. Kaushik and W. Gropp
   Optimizing Sparse Matrix-Vector Operations on Scalar and Vector Processors
3. PETSc documentation
   http://www.mcs.anl.gov/petsc/documentation/index.html
4. PETSc tutorial
   http://www.mcs.anl.gov/petsc/documentation/tutorials/index.html
5. Stream benchmark
   http://www.cs.virginia.edu/stream/