Introduction to PETSc

Performance

Loïc Gouarin

Laboratoire de Mathématiques d’Orsay

May 13-15, 2013
Outline

1. Introduction

2. Two examples of performance measurement
1 Introduction

2 Two examples of performance measurement
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.
1 Introduction

2 Two examples of performance measurement
In the most cases, when you use linear and nonlinear solvers, you perform

- AXPY vector operations
- Matrix-vector products for sparse matrices
- Dot products
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. \[ \alpha_j := \frac{(r_j, r_j)}{(A p_j, p_j)} \]
4. \[ x_{j+1} := x_j + \alpha_j p_j \]
5. \[ r_{j+1} := r_j - \alpha_j A p_j \]
6. \[ \beta_j := \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)} \]
7. \[ p_{j+1} := r_{j+1} + \beta_j p_j \]
8. EndDo

Figure: Iterative Methods for Sparse Linear Systems - Youcef Saad
Let $x$ and $y$ two vectors of size $n$. We want to compute

$$y = y + \alpha x$$

with $\alpha = 1$. 
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 $\Rightarrow 2n$ flops

- **Memory access**
  
  2 READ and 1 WRITE $\Rightarrow 3nb$ bytes

where $b$ is the size of the data (float: 4 bytes, double: 8 bytes, ...).
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 $\Rightarrow 2n$ flops

- **Memory access**

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

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

For $n \gg 1$, $3nb \gg 2n \Rightarrow$ memory-bound problem
Theoretical peak FLOPS

On my laptop \( \rightarrow \) Intel Core 2 (2.53GHz)

- 4 cores
- sse

\[
P_{\text{peak}} = 2.53 \times 2 = 5.06 \text{Gflops} \text{ for one core.}
\]

\[
P_{\text{peak}_4} = 2.53 \times 2 \times 4 = 20.24 \text{Gflops} \text{ for 4 cores.}
\]
## Memory bandwidth

### Stream2 benchmark

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Code</th>
<th>Bytes/ite read</th>
<th>Bytes/ite written</th>
<th>FLOPS/iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL</td>
<td>$a(i) = q$</td>
<td>0 (+8)</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>COPY</td>
<td>$a(i) = b(i)$</td>
<td>8 (+8)</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>DAXPY</td>
<td>$a(i) = a(i) + qb(i)$</td>
<td>16</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>SUM</td>
<td>$sum = sum + a(i)$</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

[http://www.cs.virginia.edu/stream/stream2/]
Memory bandwidth

DAXPY results

![Graph showing memory bandwidth vs. size of each array (8 byte elements)]
For 2 operations, we need $3b$ memory access.
So to have $P_{\text{peak}}$, the memory bandwidth required is

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

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

$$B_{\text{peak}} = 5.1 \, \text{GB/s}$$

So the maximum performance is

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

\[
\begin{align*}
n &= 5 \\
nnz &= 14 \\
row\_ptr &= \{0, 2, 5, 8, 11, 14\} \\
\text{col\_ind} &= \{0, 1, 0, 1, 2, 1, 2, 4, 2, 3, 4, 0, 3, 4\} \\
\text{val} &= \{1, -2, -4, 1, -2, 2, 5, 2, 1, -3, 3, 8, 2, 1\}
\end{align*}
\]
Matrix-vector product

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

```c
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.

```c
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**
  \[ 2\text{nnz flops} \]

- **Memory access**
  \[ 4n + 4\text{nnz} + 8\text{nnz} + 8n + 8n \text{ Bytes} \]
The memory bandwidth required is

\[ B_{\text{req}} = \frac{10n + 6nnz}{nnz} P_{\text{peak}} \text{GFlops} \]

Recall that the best memory bandwidth on my laptop for sufficiently large values of \( n \) is

\[ B_{\text{peak}} = 5.1 \text{GB/s} \]

Then the maximum performance is

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

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

\[ P_{\text{max}} = 637.5 \text{MFlops/s} \]
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.

```c
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);
```
By default, PETSc provides one profile listing. You can add logging stage

```c
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 your application: matrix creation, solver, ...
References

1. Matthew Knepley’s tutorial (Serial Performance Part)

2. D. Kaushik and W. Gropp
   Optimizing Sparse Matrix-Vector Operations on Scalar and Vector Processors

3. PETSc documentation

4. PETSc tutorial

5. Stream benchmark
   http://www.cs.virginia.edu/stream/