Abstract. In this report we describe a prototype implementation of a meshless method using the MPI-based Portable, Extensible Toolkit for Scientific Computation. In particular we partition the meshless point distribution, and then assemble and solve the interpolation matrix. We observe improved performance when the data is partitioned and excellent scaling results as the number of nodes are increased.

1. Meshless methods

Mesh generation is an arguably computationally expensive and non-robust computation, which also happen to be necessary for finite element methods. It has been observed that advancing front point generation is an order of magnitude faster than advancing front grid generation\textsuperscript{1}. On the other hand, meshless methods do not require a mesh to be computed and are otherwise capable of approximating solutions to the same PDEs as finite element methods with similar error bounds and convergence orders. Therefore, meshless methods have the potential to be useful alternatives to finite element methods. Despite this, their use amongst computational scientists is relatively rare compared with the use of finite element methods. One reason for this may be the relative lack of widely available high-performance, parallel, meshless methods codes, in particular for distributed memory architectures. This project takes a step in the direction of providing such a code, by implementing a representative meshless method, radial basis function interpolation, on a distributed memory MPI cluster.

Implementation of this meshless method requires dealing with problems typical of implementing most meshless methods on a distributed memory architecture. This includes constructing and solving a matrix based on a meshless point distribution, subsets of which are stored different nodes. In order to compute the entries relating points on different nodes, ghost points must be communicated. Because such internode communication is expensive, partitioning to minimize internode connectivity, is essential.

The problem that RBF interpolation solves is as follows. Given a set of values $v_1 \ldots v_N \in \mathbb{R}$ and disjoint points $x_1 \ldots x_N \in \mathbb{R}^d$, solve for coefficients $\alpha_1 \ldots \alpha_N \in \mathbb{R}$ so that the interpolant

$$s(x) = \sum_{j=1}^{N} \alpha_j \Phi(x - x_k)$$

satisfies

$$s(x_j) = v_j$$

\textsuperscript{1}Löhner, Oñate, An advancing front point generation technique. 1998
for \( j = 1 \ldots N \). If the chosen radial basis function \( \Phi : \mathbb{R}^d \rightarrow \mathbb{R} \) is positive definite then this defines a positive definite linear system of equations represented by the matrix \( A \in \mathbb{R}^{N \times N} \) which is defined by

\[
A_{jk} = \Phi(x_j - x_k)
\]

with data vector \( b = (v_1 \ldots v_N) \) and unknown solution vector \( x = (\alpha_1 \ldots \alpha_N) \). Other meshless methods involve a similar matrix in that its structure will be the same, but which is defined slightly differently to enforce differential equations.

## 2. Our PETSc-based implementation

The Portable, Extensible Toolkit for Scientific Computation\(^2\) (PETSc) is an MPI-based library which includes modules dealing with vectors, matrices, partitioning, regular grids, preconditioners, linear and nonlinear solvers, multigrid techniques, and time-stepping. PETSc also provides a framework for debugging and profiling parallel codes. This library has been successfully used to write a number of high performance codes\(^3\). PETSc is written in C, with interfaces available for C, C++, Fortran 77/90, and Python. However, the Python interface is immature and the Fortran interface is less complete than the C interface and often requires inconvenient workarounds to deal with shortcomings of Fortran 77, even when using Fortran 90.

A meshless methods code will ultimately require most of the features provided by PETSc. Following the principle of code reuse, we have chosen to build off of PETSc. The problem the code solves can be described as follows. The input required is a meshless point distribution and data values, along with implicitly connectivity (not defined by a mesh, rather the radius of influence of each point). We have implemented, with the aid of PETSc, three major computations required of implementing meshless methods on MPI-based clusters: repartitioning to minimize internode connectivity between points, matrix assembly, and solving the system of equations using an iterative solver.

### 2.1. Repartitioning

The scattered points have connectivity implicitly defined. In order to minimize data transfer between nodes of the cluster, it is required to minimize the amount of internode connectivity between points on different processor. PETSc provides partitioning capabilities. Given an initial partitioning of the points, one computes an adjacency matrix. PETSc’s partitioner will then compute an index set representing a redistribution of the points, so that inter-node connectivity is minimized. With this index set the points are scattered using PETSc’s vector scatter to their new processor. Before this is done, indexes must be updated in order to reflect the new distribution of points. In our code, this entire operation is implemented by the function Partition.

### 2.2. Computing weights

PETSc provides a Matrix module, capable of representing matrices stored in many formats. Due to the sparse structure of our matrix we chose to use PETSc’s sparse AIJ matrix format. It is emphasized in PETSc’s documentation that for efficiency storage should be preallocated for the matrix, by


specifying the number of nonzero columns in each row. This avoids reallocating
internal memory as entries are added. We have taken this more efficient approach.

Each processor stores a subset of the points in a PETSc vector. PETSc vectors
automatically distribute the data amongst processors which can easily be scattered
to other vectors asynchronously using PETSc’s scatter functionality. This is extremely useful for dealing with ghost points. Ghost points are points connected
to locally stored points which are not themselves stored locally. We use PETSc’s
scatter functionality to communicate ghost points from the processor where they
are locally stored into a ghost point vector. Before initializing this scatter, we ag-
gregate the indexes of all ghost points, required per processor, into one list without
duplicates, in order to minimize redundant communication. The scatter is then
initialized and the code proceeds.

While the scatter occurs in the background, matrix entries are computed corre-
sponding to locally stored points. With a good partitioning and data distribution,
most entries will only require points stored on the same processor. Once these en-
tries are computed, and the ghost points scattering has completed, entries involving
ghost points are computed. Throughout computing matrix entries, local points are
indexed using both their global and local index, and ghost points are indexed using
both their global and local ghost vector index. In order to map between these dif-
ferent types of indexing as required, we used PETSc’s index mapping, which allows
us to specify the mapping once, and from then on, PETSc will map indexes one
representation to the other when requested. Once all entries have been computed,
PETSc’s MatrixAssembly routines then finalize the matrix assembly.

PETSc’s linear equations solver module (KSP) is then used to solve the inter-
polation system. This module is very powerful, providing a uniform interface to a
number of parallel preconditioners and linear solvers where the interpolation matrix
and data and solution vectors are distributed across nodes. To use this solver one
specifies the system matrix, data vector, and unknown solution vector. Beyond this one
can specify additional options. For example, because our matrix is in addition
positive definite and symmetric, we used the conjugate gradients method, which is
done by simply calling the function KSPSetType with parameter KSPCG. We also
specify a tolerance for the norm of the residual. The system of equations is then
solved in parallel using the KSPSolve function.

The solution’s correctness was verified by comparing it with the solution provided
by an existing serial code.

3. Performance

We tested PETSc using its built-in performance logger, which can be used by ac-
cessed by running a PETSc code with the -log_summary command-line argument.
We tested our code with a scattered point distribution consisting of $2^{16}$ points,
where each point is implicitly connected to approximately 20 to 50 points. Table
1 reports timing measurements. The PETSc FAQ states that “You will not see
speed-up when you using the second processor. This is because the speed of sparse
matrix computations is almost totally determined by the speed of the memory, not
the speed of the CPU.” This explains the nonmonotonic performance scaling, since
each node on the cluster runs two instances of the parallel job, a setup which is
beyond our control. With this in mind, we have achieved excellent parallel scaling.

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
\# Nodes & Time (seconds) \\
\hline
1 & 1.822 \\
2 & 1.914 \\
4 & 0.944 \\
8 & 0.503 \\
\hline
\end{tabular}
\caption{Interpolation matrix assembly and solution performance measurements, recorded by PETSc’s performance logger.}
\end{figure}

In particular going from one node to four nodes more than doubles performance and from one node to eight nodes, nearly quadruples performance.

4. **Conclusions**

We have written a meshless method code for distributed memory MPI clusters. Solving such a challenging problem was made possible by effectively exploiting the code PETSc, which already deals with many of the problems involved in meshless methods. Our implementation is scalable, since all data is distributed amongst nodes. By partitioning performance increases due to less communication between processors.