NEKBONE: Thermal Hydraulics mini-application

Nekbone Release 3.1

October 10, 2013

Contents

Chapter 1. Introduction to Nekbone

NEKBONE Release 3.1

Nekbone captures the basic structure and user interface of the extensive Nek5000 software. Nek5000 is a high order, incompressible Navier-Stokes solver based on the spectral element method. It has a wide range of applications and intricate customizations available to users. Nekbone, on the other hand, solves a Helmholtz equation in a box, using the spectral element method. It is pared down to include only the necessary features to compile, run, and solve the applications found in the `test/` directory. Since almost all practical applications are in the three dimensional space, the solver is set to work with three dimensional geometries as default. Nekbone solves a standard Poisson equation using a conjugate gradient iteration with a simple or spectral element multigrid preconditioner on a block or linear geometry (parameters are set within the test directory of the simulation). Nekbone exposes the principal computational kernel to reveal the essential elements of the algorithmic-architectural coupling that is pertinent to Nek5000.

More information about nekbone can be found on the CESAR website:
https://cesar.mcs.anl.gov/content/software/thermal_hydraulics
or by contacting one of the developers.

```
Paul Fischer    :  fischer@mcs.anl.gov
Katherine Heisey:  heisey@mcs.anl.gov
```

This document contains the quick start guide, an overview of the more detailed parameters available to the user, a more detailed basis of the representation of nekbone and Nek5000, and an overview of the provided examples

Chapter 2. Getting Started

Nekbone requires the use of a F77 and C compiler. The currently tested and supported compilers are IBM, Intel, PGI Portland, and GNU gfortran, although others may be used.

2.1 Setup

For the latest version of the nekbone code, please visit
https://cesar.mcs.anl.gov/content/software/thermal_hydraulics

After downloading the nekbone tarball, it can be unzipped and extracted in one step, if using the linux-package, GNU tar commands:

```
tar -zxvf nekbone-3.1.tgz
```

This will create a nekbone-3.1directory populated with the source and test example directories.

Nekbone's test directory`(nekbone-3.1/test)` includes the example cases for running Nekbone..

Nekbone's source code is found in `nekbone-3.1/src/`.

All nekbone test or example cases must have a SIZE file and a data.rea file, unless running a test with only the communication/platform profiler. (example in nekbone-3.1/test/nek_comm/) Every test is compiled and linked with a makenek script, found in each example in nekbone-3.1/test/. This script performs a series of checks on the setup environment and compiler flags before compiling the source code using, makefile.template to create the makefile.

For more details on these examples and how to modify them, see section 5.1

2.2 Running A First Example

Change to the main example directory:

```
cd nekbone-3.1/test/example1
```

Check that the makenek script points to the correct source directory and edit it, if needed. The default is set to:

```
SOURCE_ROOT='$HOME/nekbone-3.1/src'
```

Check that the compiler is set as desired. The default compiler is set to a mpi wrapper for F77 and C. If needed, change the `F77` and `CC` parameters in the makenek script found in `nekbone-3.1/test/example1/`

Compile the code using the makenek script to build and link: `./makenek ex1` More details on the makenek script and how to modify it are found in Section 5.2.

A successful compilation of the code should result with this message printed to the screen:

```
##############################################
#          Compilation successful!           #
##############################################
```

and a nekbone executable in the `test/example1/` directory.

Running serial To run the case in serial:
`./nekbone ex1`

Running in Parallel To run the case in parallel the user can use the script provided, nekpmpi. This script will redirect the stdout to a logfile named according to the example case and the number of processors ran on. The user must supply the name of the example and the number of processors to use, i.e:
`./nekpmpi ex1 4`
would run ex1 on 4 processors.

$**$ NOTE: to run the application in parallel, one must be sure that the parameters set in the SIZE file accommodate the desired run parameters(specifically, lp and $lelt$). See section 3.1 for more details on these parameters.

To interpret the output, please see Section 5.3

Cleaning up To clean up the source and test directory, removing the .o files, use:
`./makenek clean;`

Chapter 3. Parameters in Nekbone

To run an application, much like the standard Nek5000 examples, the user must run their experimental cases in a separate directory from where the code is stored. Each case ran with the nekbone code must have a SIZE file and a data.rea file in the running directory.

3.1 SIZE File

The SIZE file contains some basic parameters needed to create the mesh and control the parameter space. Below is a brief description of the parameters found in SIZE and how they can be changed to fit the user's needs. Most of the SIZE parameters are representative of the local processor counts as opposed to a global element representation.

$ldim$ this is the dimension of the example. The code is written to work with three dimensions. Changing this parameter would produce unexpected results and is not recommended.

$lx1, ly1, lz1$ without being recompiled, this is the maximum polynomial degree set as $N = lx1 - 1$, where N is the polynomial degree. It can be any number, even or odd, that is greater than or equal to two. On some machine platforms, an advantage has been seen when using even numbers. However, on others there has been no evidence that either should be preferred. The parameters $lx1$, $ly1$, and $lz1$ should always be equal.

lp the maximum number of processors that can be used without recompiling the code. This parameter should be changed to reflect the MAXIMUM number of processors the user plans to run with.

$lelt$ the maximum number of elements per processor that can be ran without recompiling the code. This should reflect the MAXIMUM number of elements per processor.

$lelg$ the total number of elements in the run. This is set to be $(lp \times lelt)$ and should not be changed. The code is currently set to find the best configuration across processors using this total number of elements in each dimension space, or can be configured by the user in the data.rea file.

ldimt this parameter is used in the include parameter files and should be kept as is.

common/dimn/ is the common block containing some of the most used variables in the code. Most are initialized in the beginning of the code and are equated to their counterparts named similarly. (i.e., $ndim = ldim$; $nx1 = lx1$; ect..) In general, these are the case specific parameters, not the bounding sizes.

3.1.1 Impact of Parameters

The parameters set in the SIZE file define the problem space to be evaluated. As stated above, *lelt* determines the number of total elements(per proc.) in the geometry where as $lx1, ly1, lz1$ define the polynomial order. As the figure 3.1 shows the polynomial order really enriches the geometry by increasing the total number of gridpoints.

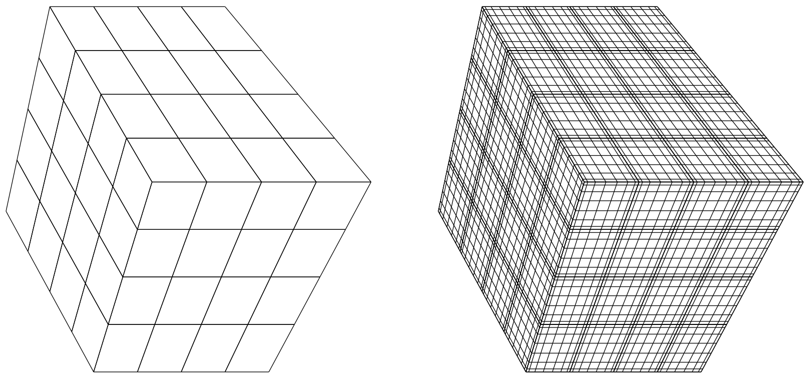

Figure 3.1: The role of $lx1$. The right geometry has no $lx1$ defined whereas the left has $lx1, ly1, lz1$ set to 7.

3.2 data.rea File

Along with the SIZE file, the data.rea file provides the user with a few parameters to be changed at runtime. This will allow users to alter certain variables without having to recompile the code.
EVERY EXAMPLE must have a data.rea file with these variables set:

ifbrick This is the logical switch used to determine a brick geometry or a linear block of elements. Setting `.true. = ifbrick` will allow the code to determine the ideal 3-D configuration of nelt elements and np processors. Setting `.false. = ifbrick` will trigger the linear geometry. The linear geometry yields itself to an optimal communication pattern since each element only needs to communicate to 2 other elements on either side of it. (Excluding the ends, which would only have one neighbor) The brick configuration has a more realistic communication pattern where the interior elements need to communicate with 8 neighbors. Since the element counts are configured on a per processor basis, certain values of nelt will produce a linear geometry, despite the ifbrick flag being set to true. Prime numbers or geometries that don't lend to a X-by-Y-by-Z decomposition will default to the linear bring of NELT-by-1-by-1.

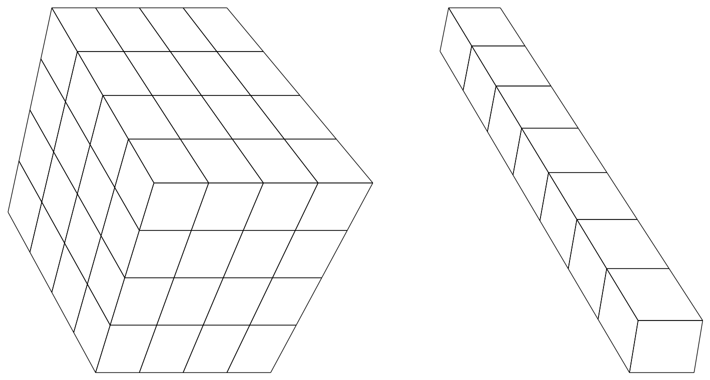

Figure 3.2: When `.ifbrick.` parameter is set to false a linear geometry is created (left) and when set to true, a 3-D brick of elements is created(right)

$iel0, ielN, step$ These three values are read in by nekbone and will control the range of elements to be evaluated, per processor. Nekbone will run a battery of tests starting with $iel0$ through $ielN$ elements per process, jumping by $step$. Thus, setting $iel0$ to 1, $ielN$ to $lelt$, and $step$ to 1 will loop through tests with 1 element per processor to $lelt$ elements per processor, as set in the SIZE file. If desiring a test that only runs a single Poisson solve on a specific element count, set $iel0 = ielN$. These values can be changed at runtime and do not require nekbone to be recompiled as long as $ielN <= lelt$. It is important to note that $step$ is the added to the number of elements, per process, and will terminate tests once $step{+}number\_of\_elements > ielN$. Thus, the last test ran may not be of size $ielN$, but of a smaller element count based on the addition of $step$ to the previous count.

$nx0, nxN, step$ These parameters are used to vary the polynomial order of a run. Currently, these are defaulted to be the $lx1$ value set in the SIZE file. These parameters will be read from the data.rea file and control the range of polynomial order. The polynomial order is set to be

$nx1-1$ where $nx1$ is set according to $nx0$, nxN, and $step$. Setting $nx0 = 2$ and $nxN = lx1$ will run a series of tests from 2 through $lx1$ giving the full scope of polynomial ordering up to the maximum value set in SIZE. The default setup sets $nx0 = nxN = lx1$, therefore only running with a constant $nx1$ value equal to what is set in the SIZE file. Varying the polynomial order will change the computational complexity by increasing the number of grid points per element. Typical values span anywhere from 4 to 14, although much larger values have been explored. When using the multigrid preconditioner, $nx0$ must be at least 4. Anything smaller will result in an error message and overriding the starting polynomial value to 4.

npx, npy, npz When $ifbrick$ is set to true, the user can also control the distribution of processors in the simulation. npx, npy, npz represent the decomposition of the processors in the respective Cartesian coordinates. If $npx \times npy \times npz$ does not equal the number of processors being used, np, nekbone will find the correct configuration of np processors. Thus, setting npx, npy, npz to zero will result in allowing nekbone to calculate the processor decomposition.

mx, my, mz Similarly, the user can set the local element distribution using the data.rea parameters, mx, my, mz. If $mx \times my \times mz$ does not equal the $nelt$ for the current test, nekbone will calculate the best cubic decomposition for the user.
NOTE: When running test with varying $nelt$ values (as set by parameters $iel0, ielN, step$ in data.rea) the mx, my, mz decomposition will only be applicable for the first test. The proceeding tests will be configured by nekbone's decomposition routines.

Chapter 4. The Nekbone Code: Default Scheme

Nekbone solves the Helmholtz equation in a box using the spectral element method. It partitions the computational domain into high-order quadrilateral elements. Based on the number of elements, number of processors, and the parameters of a test run, Nekbone creates a decomposition that is either a 1-dimensional array of 3D elements, or a 3-dimensional box of 3D elements. It evaluates a Poisson equation on every time step iteration and provides an estimate of realizable flops, as well as inter node latency and bandwidth measures.

4.1 The Default Setup

Boundary Conditions In order to ensure Dirichlet boundary conditions, a mask is applied in each conjugate gradient iteration. For simplicity this mask zeros out all faces of the global geometry, enforcing Dirichlet conditions everywhere. This maintains a solvable code while not complicating it with an extravagant masking mechanism.

Preconditioning Set in the makenek script, nekbone can be ran with a multigird preconditioning step or a simple application of the identity matrix. The multigrid preconditioner is an iterative method based on the idea that a coarser mesh representation will expose highly oscillatory components of the error that would otherwise seem insignificant on the original mesh. It has two fundamental aspects, the local relaxation (or smoother) and the coarse-grid correction. These are applied in a recursive fashion to achieve a multilevel preconditioner. Nekbone is set to have 2 coarsening steps, arriving at the coarsest level of mesh representation, 1 grid point per spectral element. At this level, there is a coarse grid correction calculated and then applied to the coarse solution. This correction is then propagated up through the mesh representations until it is finally applied to the original, fine grid mesh. The multigrid precondition speeds up convergence of the conjugate iteration scheme. It also more closely represents the methods that are used in current application codes. Obviously, this comes at an increase in memory and communication overhead, as well as an increased algorithmic complexity to the mini-application as a whole. To help curb this added complexity, we have applied an elementary coarse-grid solution as opposed to a more sophisticated scheme. Nekbone still upholds the option of running without this preconditioning step and instead implementing the original identity matrix as was done in previous nekbone releases. The application of the Identity matrix to the initial solution, otherwise known as a copy of the initial guess. Using the multigrid precondition rapidly increases convergence and is a realistic method used by actual, full application codes; however, it will require more memory and communication overhead. To run with the multigrid turned on, simply add MGRID to the PPLIST= variables set in the makenek script of the example directory.

Platform Timing Tests The default application of this code has the capability to run a battery of tests useful in profiling the platform and basic communication structure of the code. The first set of tests are called in driver.f by:

```
call platform_timer(iverbose)
```

The variable, iverbose, controls how much information is sent to standard output and can be flagged with a 0, for not verbose, or a 1 for verbose. These tests include ping-pong tests and all-reduce tests to give relevant information about the platform being ran on. Currently, this call is commented out to allow for short test runs with the main example case `nekbone-3.1/test/example1`. However, the included example `nekbone-3.1/test/nek_com` is set to run only the platform timer and exits after reporting the communication data.

Poisson Evaluation After the platform timing tests, the Poisson equation evaluations begin. An iterative conjugate gradient solve, with a precondioning step, is performed on an increasing number of elements per process from $iel0$ to $ielN$ (jumping by $step$), set in the data.rea file. (See 3.2) The principle kernel of the code is the $\underline{w} = A * \underline{p}$ routine with has many opportunities for optimization. Essentially, the bulk of this work is done through matrix-vector products. These $\underline{w} = A * \underline{p}$ evaluations are done on the local elements on each processor. To update across all processors, a nearest neighbor communication must be executed. The conjugate gradient evaluation iterates for a determined number of iterations, $niter$, set in driver.f.

Counting FLOPS Inside the conjugate solver, the flops are counted and timed for further analysis as the problem size grows. This counter is output to the logfile for each problem size as:

```
nelt= nelt, np= np, nx1= nx1, elements=nelt*np
Tot MFlops= mflops*np , MFlops = mflops
Setup Flop= flops_a , Solver Flop= flop_cg
Solve Time= time
```

where,for MPI processes running on rank 0,
nelt is the number of elements
np is the number of processors
nx1 is the value of nx1, polynomial order
mflops is the total_number of flops divide by time_spent in solver
flops_a is the operations spent in the Ax=b routines
flops_cg is the operation count spend in the conjugate gradient
time is the total time spend in the solver

The average number of flops per test run is output at the end of the final test.

```
Avg MFlops = avgFlop
```

This is the total flops averaged over the number of individual tests ran in a single submission.

Calculating Bandwidth Finally, the bandwidth of processors *np* can be tested with a call to:

```
call xfer(np,cr_h)
```

Here an array of increasing size is exchanged and timed across processors, averaged over 50 exchanges. This gives an idea of bisection bandwidth capacity of a range of data sizes. It is essentially testing the rate of message transfers with increasing sized messages, over the total number of processors. The default setting has this call commented out to speed up the overall time spent in any nekbone case.

Chapter 5. Details on Running and Editing Nekbone Examples

Nekbone's test directory`(nekbone-3.1/test)` includes the example cases for running nekbone. Most of these examples can be edited and customized to test various aspects of the application.

5.1 Editing the Nekbone Test Examples

Number of elements

These examples will run a battery of problems in a single submission. Each problem increases in element count by the total number of processors being ran, times *step*, the variable set to control the incremental increase of each test iteration. Thus, the problem sizes can range from one element per process to *lelt* elements per process, where *lelt* is set in the SIZE file. This can be changed to range from any beginning and ending number of elements per processor by changing the parameters $iel0$ and $ielN$ in the data.rea file. Similarly, *step* can be set to control the jump of elements per process by setting it to 1 or any integer $< ielN$. As described in 3.2, these parameters control the range of tests to be ran as $iel0$ though $ielN$ as long as $iel0 > 0$ and $ielN <= lelt$. The default sets $iel0$ to 1 and $ielN$ to $lelt$, thus running a total of $lelt$ tests, increasing by one element per test.

Varying Polynomial Order

The default setup of nekbone runs with a set polynomial order equal to $lx1$ as set in the SIZE file (see Section 3.1). This can be configured to run a range of increasing polynomial orders by setting the parameters $nx0 >= 2$ and $nxN <= lx1$, and editing *step* to the desired increment. See 3.2 for more details.

Specifying the processor decomposition

The default variables, npx, npy, npz are set allow nekbone to find the best decomposition of the number of processors, np. This can be edited in the data.rea file, to establish a specific arrangement of the processors. However, if the user provided decomposition does not add up to the total number of processors, nekbone will find the best 3D distribution. This allow the user to create a processor breakdown that reflects a 2D processor map, instead of the 3D one set with $ifbrick = true$ or the linear one set with $ifbrick = false$.

For example, 8 processors would usually be decomposed into a 2-by-2-by-2 mapping, but the user could change the data.rea file to create a 4-by-2-by-1 map.

Specifying the local element decomposition

Similar to the processor distribution, the default variables, mx, my, mz are set allow nekbone to find the best decomposition of the elements per process, $nelt$. This can be edited in the data.rea file, to establish a specific arrangement of the local elements. However, if the user provided decomposition does not add up to the total $nelt$, nekbone will find the best 3D distribution. Like the processor mapping above, these parameters allow the user some flexibility to create element structures that are not just the 1D or 3D decomposition. However, since the parameters $iel0, ielN$ allow for a varying number of element counts to be evaluated in a single submission, the user provided, local element decomposition will only be read in on the first run, $iel0$ elements per process. For test ran after, mx, my, mz will no longer equal $nelt$ and nekbone will find the decomposition.

Naming Examples

In the current set up, the name of the example is not important and is not used. In future revisions, this might become integral to the code. However, as a basic set up, we have used the SIZE and data.rea parameters to specify the exact specifications. No mesh data or input data is read in besides these two files.

Geometry

Using the logical variable, $ifbrick$, found in data.rea, the user can control whether the geometry is set to be a brick or just a single line of elements. $ifbrick$ in this example is set to false, resulting a the optimal communication pattern that a linear geometry lends itself to.

In any single test run, the total degrees of freedom are
$dof = np \times nx1^3 \times nelt \leq lp \times lx1^3 \times lelt.$

number of CG iterations

The conjugate gradient solver is set to run for a maximum number of iterations, niter. niter is set in `src/drive.f` and is can be increased as the degrees of freedom increase in the example. A lower niter value may result in non-converging results simply due not being allowed to iterate in the solver long enough for the degrees of freedom to be resolved.

5.2 Compiling Nekbone

Nekbone is compiled by running the provided script, `makenek`. Makenek allows the user to set the compiler, any compiler flags, optimization flags, and other preprocessing flags.

SOURCE_ROOT One of the important variables that is defined in the script is the source directory path, `SOURCE_ROOT=`. This should be set to the path to the source code. Since the tests are all ran from their own directory, this path can be locally defined as
`../../src`
or more globably as the path from the user's HOME/ directory. As default, the path is set to
`$HOME/nekbone-3.1/src`
which assumes that the tarball was downloaded and unzipped in the HOME/ directory.

F77 and CC F77 and CC are the compilers to be used. Nekbone has been tested with GNU's gfortran, PGI Portland, INTEL and a few others. Both serial and parallel version have been used. The standard, mpif77 and mpicc are default in the test directory.

PPLIST PPLIST sets pre-defined pre-processor symbols that are used within nekbone. The current options are:

- BG - Currently, setting this variable to BG will enable some optimizations specific to Blue Gene P platforms.
- NEKCOMM - When running the communication-only example, `test/nek_comm`, PPLIST must include NEKCOMM to compile the correct files.
- NEKDLAY - To compile the correct files for the nek delay example, `test/nek_delay`. This cannot be combined with NEKCOMM.
- MGRID - to turn on and compile the multigrid preconditioner.
- ? - this will give a list of acceptable symbols nekbone can be configured with

IFMPI Uncommenting this variable sets IFMPI to false. As the name implies, this would turn off MPI communication within nekbone and enable a serial run. This should be toggled to false when using a serial compiler or when wanting to run without MPI enabled.

G The *G* variable is for any compiler flags the user wants to include. A common setting is compiling with debugging turned on by setting G = "-g". For PGI Portland serial compilers, adding -Ktrap=fp will cause the test to exit when encountering any NaN values.

Optimization Flags General optimization flags can be specified by setting the

OPT_FLAGS_STD variable as desired. This will set the optimization level for a majority of the source files. If this is not specified, the code is compiled with $-O2$ and with $-O0$ when in debugging mode.

OPT_FLAGS_MAG is used to set the highest level of optimization, which is used on some of the of the more intricate files. If this variable is undefined, these files with be compiled with $-O3$ and $-O0$ when in debugging mode.

Once the variables are defined as desired, running makenek in the test example directory:

```
./makenek name_of_test
```

will compile and link the code to be ran. See section 2 for more details on running the example provided in `nekbone-3.1/test/example`.

5.3 Understanding the Output

5.3.1 Element and Processor Distribution

According to the logical, ifbrick, set in data.rea, nekbone will distribute the local elements in a linear or brick geometry. This element distribution will be output to stdout as:

```
Processor Distribution:  npx,npy,npz
Element Distribution:  nelx,nely,nelz
Local Element Distribution:  mx,my,mz
```

where the processor configuration will reflect the total processor distribution, the element distribution will be the total element counts in the x,y,z direction, and the local element counts will be the element configuration on each processor. Thus, the total elements in each direction will equal the processor count multiplied by the local element count. ($npx \times mx$) This distribution can be edited by parameters in the data.rea file. See 3.2 and 3.2 for more details.

5.3.2 Platform timer Results

When the platform_timer(ivrb) is turned on, the result of all platform tests will be at the beginning of the logfile. This includes all_reduce times, varying times of different matrix- matrix product routines, and ping pong tests done on the platform ran.

The first set of data in the platform timer results is the mxm routine tests. Starting with a 1-by-1 matrix and increasing to a 16-by-16 matrix, the various matrix-matrix routines are tested based on flops, speed, and peak harmonic.

Labeled with 'gop', the all_reduce test results are reported next. Starting with message sizes of 1 and increasing to 100,000 words, the all reduce times are reported as
`np nwds tmgs tpwd "gop"`
where:
np is the number of processors
nwds is the number of words in the message
tmgs is the total time to communicate this message across processors
tpwd is the time per word.

Similar results are reported with "gp2" which is a Nek written all_reduce implementation using a fan-in/fan-out method.

Finally the ping-pong test results are output with the label "pg". These tests time how long it takes for a message to travel between two nodes. When the node count is over 256, the sampling of nodes is spread out among the remaining nodes to adequately span the communication topology while only running 512 total ping pong tests.

The stdout for the ping-pong tests is:
`nodeb np nloop nwds tmgs tpwd`
where:
nodeb is the node number that node 0 is communicating with
np is the total number of processors
nloop is the number of iterations the ping-pong test used
nwds is the number of words in the message sent between nodes
tmgs is the total message time
tpwd is the time per word

After each node0 -¿ nodeX pair finishes its ping pong test, an approximate alpha and beta value is reported.

5.3.3 Conjugate Gradient & Flop Counts

Nekbone writes to stdout the results of each conjugate gradient sequence on increasing problem size.

At the beginning of each sequence, nekbone prints:
`cg:  iter rnorm`
where iter should be 0, since the test is just beginning.

After *niter* iterations (set within `src/drive.f` of the nekbone source code), a summary of the convergence is printed to stdout.

```
cg:iter rnorm alpha beta pap
```

The current configuration will run the conjugate gradient iterations for a set number of iterations, *niter*, regardless if the solution has converged beyond the set tolerance. The reported *rnorm* value is the error between the 'correct solution' and the found solution.

After the conjugate gradient sequence is completed, the total flop count is printed to the screen.

```
'nelt , np nx1 , elements'
'Tot MFlops , MFlops'
'Setup Flop , Solver Flop '
'Solve Time '
```

More detail is given in Section 4.1. Since the default implementation of nekbone is set to run increasing elements per processor, *np* remains constant and *nelt* should increase from 1 to *lelt* (set in SIZE) or from $iel0$ to $ielN$ (set in data.rea).

5.3.4 Bandwidth Test

If the bandwidth bisection test is turned on, nekbone will print the results of the gather-scatter routines using the crystal router exchanges done on an increasing number of points per process.

```
np npts npoints etime "bandwidth"
```

Where
np is the total number of processors
npts is the points per process exchanged
npoints is the total number of points in this test ($np \times npts$)
etime is the average time it took to exchange these points across processors.

This will test the rate of message transfers with increasing sized messages of the total number of processes.

Chapter 6. Overview of Included Nekbone Examples

The nekbone test directory includes several examples to illustrate the various approaches applicable within nekbone.

6.1 The basic nekbone example, A template

The primary nekbone example is found in `nekbone-3.1/test/example1`. This is the example used in the quick start, 2 and is considered the baseline example of the nekbone mini application.

All of the default setup and editing remarks , found in 4.1, 5.1, and 5.2 apply to this example.

There is a considerable amount of flexibility within this example to modify the problem based on the parameters found in the SIZE and data.rea file. (See section 3.1)

6.2 The simplest example, example2

To run a clean, all parameters set at default, test, `nekbonei-3.1/test/example2` has been provided. This example will run one test with a set number of elements per processor and a set polynomial order. Further, it will run without the multigrid preconditioner and without user-provided decompositions of the processor counts and the elements. Some easy adjustments to the example would be to compile with MGRID in the PPLIST= variable in the makenek script. This would allow a comparison between the basic diagonal preconditioner and the multigrid preconditioner.

6.3 The more complex example, example3

Set to compile and run the multigrid preconditioner, `nekbone-3.1/test/example3` is a slightly more complex example. Besides MGRID being included in the makenek script, the data.rea parameters have been adjusted to distribute the 50 elements per processor in a more 2D decomposition. Instead of having nekbone prescribe the elements distribution, this example specifies a 25-by-2-by-1 distribution.

6.4 The Multigrid Preconditioner Example

The example is found in `nekbone-3.1/test/nek_mgrid`.

All of the default setup and editing remarks , found in 4.1, 5.1, and 5.2 apply to this example.

The makenek script includes the symbol MGRID, so the multigrid preconditioner will be enabled. Further, the data.rea file parameters have been edited to run 2 polynomial orders. The first set of tests will run with a polynomial order of 8, and the second will run the same element counts with a polynomial order of 10.

6.5 Special Platform Performance Example, nek_comm

Found in `nekbone-3.1/test/nek_comm`, this example will only perform the platform profiling tests. The reported data will detail communication measures such as the optimal matrix-vector subroutines, ping-pong tests, and all reduce times for increasing message sizes. Since this example is solely for the platform profile and does not implement any of the Poisson solver routines, the setup has been simplified. There is no data.rea file needed in this example, although one has been included for general symmetry between examples. This is an exception to the rule that all examples must use these files. All of the remarks in 5.2 pertaining to compiling nekbone examples are still relevant to this example. However, since there is no geometry to be setup, the observations in 5.1 can be overlooked.

Successful compilation of this example will result with a binary executable, nekbone, in the test directory. Note that the makefile has included a $PPLIST$ symbol, $NEKCOMM$, to trigger the inclusion of the necessary files for the communication-only tests.

6.6 Load Imbalance Example, nekdlay

By implementing a time delay based on the node id, nekbone is able to simulate a machine where only a subset of nodes have an imbalance. This example is found in `nekbone-3.1/test/nek_delay`. The delay features are triggered by including $NEKDLAY$ in the $PPLIST =$ variable in the makenek script.

The delay feature is controlled from within the driver_dlay.f file with the parameters: $tmean, tavg, rms, d\_range, int\_calls$

The delay is found by sampling from a Gaussian centered around the value, *tmean* which is set in the driver_dlay.f file. The standard deviation, *rms*, is between [0,*d_range*] using a basic rejection method. Finally, the delay is normalized so the the average is *tavg*. A resampling is done every *int_calls*.

This delay function is called before every MPI call, by every process, to simulate a processor delay.

At the end of the test, the total elapsed time is reported to demonstrate the effects of the delay on the overall time, including the setup time. In addition an increase in the overall time spent in nekbone, there should be an increase in each conjugate gradient calculation. This value is printed to stdout as part of the conjugate gradient output, explained in section 5.3.

Like the original nekbone example, nek_dlay is controlled with the parameters found in SIZE and data.rea.

Chapter 7. Nekbone & Nek5000

7.1 How Nekbone Represents Nek5000

As described above, nekbone is a conjugate gradient solver with the option for a simple or multigrid preconditioner implemented. Nek5000's temperature solve is a conjugate gradient iteration with multi-level point-Jacobi preconditioner. Any Nek5000 application that spends a majority of time in the temperature solver will very closely resemble a nekbone test ran on a large, brick element count. We have found that Nek5000 runs at parallel efficiency at ∼6,000-10,000 points per core. This means that the total degrees of freedom ($lelt \times lx1^3$) of a nekbone test should also follow this rule of thumb and one should expect to see similar results as nekbone scales to large processor counts. Also, both Nek5000 and nekbone's memory requirements scale as $lelt \times (lx1^3)$.

A Nek5000 Case with natural convection at high Rayleigh number ($Ra > 10^10$) will spend around 82% of the CPU time in the Helmholtz solve. Of this, 19% is spent in the precondition, which is not yet in nekbone. This leave 63% of the run time spent in calculations that are represented by the kernels found within nekbone. Since a principal challenge of exascale is to boost single-node performance, nekbone focuses on the main kernel in question.

7.2 MPI Communication within Nekbone

The communication kernel used in the standard Nek5000 software is the exact kernel used in this more basic code. nekbone communication is nearest-neighbor communication which is the majority of what is found in the Nek5000 application. Written primarily in C and C preprocessor, the communication routines are found in `nekbone-3.1/src/jl/`. The mini application accesses these routines to set up and exchange information across processors. The code is a parallel code, utilizing the MPI standard. Most MPI routines are employed through a wrapper found in comm_mpi.f.

7.3 Optimization Opportunities

Nekbone provides multiple levels of optimization. Since the bulk of the nekbone code focuses on the matrix-vector operations, this is a section of the code that could be highly optimized. Already, these routines have been optimized on most platforms common in the current computing resources.

The gradient kernel include 3 matrix-vector calls on the same data and the gradient-transpose kernel includes 3 matrix-matrix calls on different data to produce one output.