Performance Analysis for Mesh and Mesh-Spectral Archetype Applications

Adam Rifkin and Berna Massingill*

Computer Science 256-80
California Institute of Technology
Pasadena, California 91125
{adam,berna}@cs.caltech.edu

Abstract

This document outlines a simple method for benchmarking a parallel communication library and for using the results to model the performance of applications developed with that communication library. We use compositional performance analysis — decomposing a parallel program into its modular parts and analyzing their respective performances — to gain perspective on the performance of the whole program. This model is useful for predicting parallel program execution times for different types of program archetypes, (e.g., mesh and mesh-spectral) using communication libraries built with different message-passing schemes (e.g., Fortran M and Fortran with MPI) running on different architectures (e.g., IBM SP2 and a network of Pentium personal computers).

1 Introduction

A parallel programming archetype [Cha94, CMMM95] is an abstraction that captures the common features of a class of problems with similar computational structure and combines them with a parallelization strategy to produce a pattern of dataflow and communication. Such abstractions are useful in application development, both as a conceptual framework and as a basis for tools and techniques.

*This research is supported in part by the NSF under CRPC grant CCR-9120008. This work constitutes part of the Caltech Archetypes Project; more information is available at http://www.etext.caltech.edu/archetypes.html on the Web.
The efficiency of a parallel program can depend a great deal on how its data is decomposed and distributed or on its granularity. This paper describes a simple performance evaluation methodology that includes an analytic model for predicting the performance of parallel and distributed computations developed for multicomputer machines and networked personal computers. This analytic model can be supplemented by a simulation infrastructure for application writers to use when developing parallel programs.

These performance evaluation tools were developed with the following restricted goal in mind: We require accuracy of the analytic model and simulation infrastructure only to the extent that they suggest directions for the programmer to make the appropriate optimizations. This restricted goal sacrifices some accuracy, but makes the tools simpler and easier to use.

A programmer can use these tools to design programs with decomposition and distribution specialized to a given machine configuration. By instantiating a few architecture-based parameters, the model can be employed in the performance analysis of data-parallel applications, guiding process generation, communication, and mapping decisions.

The model is language-independent and machine-independent; it can be applied to help programmers make decisions about performance-affecting parameters as programs are ported across architectures and languages. Furthermore, the model incorporates both platform-specific and application-specific aspects, and it allows programmers to experiment with tradeoffs better than either strictly simulation-based or purely theoretical models. Finally, the model is designed to be simple.

Section 2 presents the model, related work, and our experimental methods. Section 3 and Section 4 describe applications based on the mesh archetype and the mesh-spectral archetype, for use in our performance experiments. Section 5 discusses the experiments, and Section 6 presents conclusions. Source code listings are provided in the Appendices.

2 Performance Model

Archetype-based performance models exploit commonalities in programs to simplify the process of performance analysis. The performance model in this paper, described fully by Rifkin [Rif96], is based on the ideas of extrapolation from observations, asymptotic analysis, scalability analysis, execution profiles, and data fitting, as investigated by Foster [Fos95, Chapter 3].

Since the problem of choosing the data partitioning and distribution to achieve the optimal performance is NP-complete [Rif96], we are more interested in user-guided performance evaluation tools for the refinement of parallel applications than in automatic performance prediction (for example, Fahringer’s work with $P^3T$ [Fah96a]). Since our model is likely to be used to supplement a programmer’s efforts to develop applications using archetypes, it differs from
efforts to do performance measurement for compiler optimization (as Clement and Quinn do with $C^*$ on multicomputers [CQ93]), and it differs from efforts to estimate performance statically to automate the load balancing of useful work within a program (as with Fahringer’s application of $P^3T$ [Fah96b]). Like the BSP model [Val90, GV94], our model decomposes programs into fairly large blocks; our model, however, incorporates the idea of archetypes, gaining ease of use at the expense of greater generality.

Our techniques fit in well with other methodologies for dealing with applications developed for particular architectures (for example, Brinch Hansen’s model for programming multicomputer applications [BH93a]). Archetypes frequently represent well-researched patterns or abstractions; for example, the mesh archetype [Mas96] builds on Brinch Hansen’s work on parallel cellular automata in the context of multicomputers [BH93b]. In that paper, the computational complexity of parallel cellular automata is derived and shown to be a sufficiently accurate estimator of the performance of a Laplace’s equation solver; in this paper, we provide an alternative technique of performance estimation using a combination of benchmarking and analysis that is especially suited to applications developed using archetypes. The mesh-spectral archetype [DM96] extends the parallel cellular automata model, providing row and column operations in addition to grid operations.

2.1 Methodology

A great deal of work has been done both on methods of exploiting design patterns in program development (for example, [Col89] and [GHJV95]) and on methods of solving problems on concurrent processors (for example, [FJL+88] and [BBC+93]). Archetypes [Cha94, CMMM95] were developed as design patterns with the single restricted goal of modeling one kind of pattern that is relevant in parallel programming: the pattern of the parallel computation and communication structure.

Methods of exploiting design patterns in program development begin by identifying classes of problems with similar computational structures and creating abstractions that capture the commonality. Combining a problem class’s computational structure with a parallelization strategy gives rise to a dataflow pattern and hence a communication structure. It is this combination of computational structure, parallelization strategy, and the implied pattern of dataflow and communication that we capture as a parallel programming archetype [Mas98, MC96]. For the remainder of the paper, we use the term “archetype” to refer to a parallel programming archetype.

A key question in the development of a parallel application, especially for a multicomputer or a network of computers, is the issue of data decomposition and distribution. Archetypes directly address the question of which data distributions are compatible with a problem’s computational structure. In some cases more than one data distribution is compatible with the computational structure;
in such cases, which one is chosen does not affect program correctness, but it may well affect program performance.

In this paper and related papers ([Riff96, Rif93]) we address the question of how the choice of data distribution affects performance and present a methodology for archetype-based application development, including a phase in which a correct but not necessarily efficient application is refined to improve performance by using an archetype-based performance model.

Our methodology for designing, developing, implementing, and refining an application follows the workflow schedule illustrated in Figure 1. This workflow consists of six major phases: Analysis, Application, Benchmarking, Performance Model, Simulation, and Refinement, each of which we describe in turn.

![Workflow Diagram](image)

**Figure 1:** The methodology for using and analyzing the performance of archetypes follows a simple workflow.

**Analysis.** In the Analysis phase, an appropriate archetype is chosen to help with application development. If more than one archetype could be used, our performance models can suggest which one is more efficient, as is done in Section 5.1. This phase usually starts with a problem description and/or a sequen-
tial program to be parallelized; it ends when the appropriate archetype has been chosen. Two phases of the methodology can then be worked on concurrently: development of the application (path (a) in Figure 1), and benchmarking of computation and communication routines (path (b) in Figure 1).

**Application.** In the **Application** phase, the selected archetype is used to develop an application to solve the specified problem or to parallelize the given sequential program, as follows.

First, the programmer develops an initial archetype-based version of the algorithm. This initial version is structured according to the archetype's pattern and gives an indication of the concurrency to be exploited by the archetype. Essentially, it is produced by adapting the original algorithm or program to fit the archetype pattern and “filling in the blanks” of the archetype with application-specific details. An important feature of this initial archetype-based version of the algorithm is that it can be executed sequentially; if the algorithm is deterministic, it can also be debugged sequentially.

This initial version is then transformed into an equivalent version suitable for efficient execution on the target architecture. The archetype assists in this transformation, either via guidelines to be applied manually or via automated tools. Again, the transformation can optionally be broken down into a sequence of smaller stages, and in some cases intermediate stages can be executed (and debugged) sequentially. A key aspect of this transformation process is that the transformations defined by the archetype preserve semantics and hence correctness.

The programmer then implements the efficient archetype-based version of the algorithm using a language or library suitable for the target architecture. Here again the archetype assists in this process, not only by providing suitable transformations (either manual or automated), but also by providing program skeletons and/or libraries that encapsulate details of the parallel code (for example, process creation and message-passing). A significant aspect of this step is that it is only here that the application developer must choose a particular language or library; the algorithm versions produced in the preceding steps can be expressed in any convenient notation, since the ideas are essentially language-independent.

After this implementation is tested and debugged, its performance can be evaluated and possibly improved via the remaining phases of our methodology.

**Benchmarking.** **Benchmarking** of archetype communication and computation routines can be performed during or independently of the **Application** phase. This phase collects data to be used in performance evaluation via analytic techniques and/or simulation methods.

Our performance model requires two sets of benchmarks: archetype-specific (“communication”) benchmarks and application-specific (“computational”)
benchmarks. Both sets involve measuring execution times of the relevant routines (from the archetype library for the first set, and from the application for the second) using the target architecture, language, compiler, and library. Computational benchmarks can be done on a single processor; communication benchmarks require \( N \) processors, where \( N \) is the maximum number of processors the application might use. If the programmer wants to use analysis to help choose an appropriate granularity, communication benchmarks must be done for varying numbers of processors.

Our model uses approximations of higher-level communication routines rather than approximations of low-level measurements such as latency and bandwidth. We do this for two reasons: It is a very simple method for gathering information about the potential performance of a program, and it allows developers to reuse benchmark measurements for applications using the same archetype. Recall the restricted goal: We require accuracy of the analytic model and simulation infrastructure only to the extent that they suggest directions for the programmer to make appropriate optimizations.

Once the relevant routines are benchmarked, the programmer can use the results in the Performance Model phase (path (d) in Figure 1) as well as in the Simulation phase (path (e) in Figure 1). These two phases can be done concurrently if both are desired. However, as our experiments will show, usually the Performance Model phase is sufficient (without the Simulation phase) for predicting program efficiency and for doing the corresponding performance tuning.

**Performance Model.** The Performance Model phase consists of two steps:

1. **Analysis** of the program to produce a closed-form equation involving the benchmarked quantities, and

2. **Instantiation** of the equation with the appropriate benchmarked values to give a number representing a prediction of the program's expected running time.

We call the methodology of developing the program concurrently with modeling its performance AdAPT, for Application Development using Analytic Performance Tuning [Rif96].

The Analysis segment of AdAPT involves decomposing the given program into a number of subprograms (e.g., initialization, computational loop, and termination) whose running times can be expressed in terms of the benchmark numbers. The finer the grain of decomposition and benchmarking, the more predictive we expect the model equation to be for that program.

The basis for this decompositional approach to performance modeling is a structured induction [Rif96] on the statements of the program being modeled, assuming implicit barriers between any subprograms. For example, for a program
S that consists of program $S_1$ (which may involve a number of computations and communications) composed in sequence with program $S_2$ (which also may involve a number of computations and communications), we assume an implicit barrier between $S_1$ and $S_2$:

$$S = S_1; S_2$$

For the base case in which $S_1$ and $S_2$ each consist of either a single computation, using one or more of the given processors, or a (possibly collective) communication operation also using one or more of the given processors, we can model the expected running time $R(S)$ of program $S$ as follows: $S_2$:

$$R(S) = R(S_1) + R(S_2)$$

Since $S_1$ consists of a single computation or communication, we model $R(S_1)$ as the maximum of the expected running times of that computation or communication on all of the processors. If $R_p(S_1)$ is the expected running time of $S_1$ on processor $p$, we can model $R(S_1)$ thus:

$$R(S_1) = \max_{\forall p} R_p(S_1)$$

We can model the expected running time of $R(S_2)$ similarly; as a result, we have:

$$R(S) = \max_{\forall p} R_p(S_1) + \max_{\forall p} R_p(S_2)$$

The structured induction thus allows us to compute the expected running time of a large program from the expected running times of its components. For example, having derived a closed-form equation for $R(S)$, we can use it to compute expected running of a program $T$ that consists of two executions of $R(S)$ in sequence:

$$R(T) = R(S) + R(S) = 2 \times (\max_{\forall p} R_p(S_1) + \max_{\forall p} R_p(S_2))$$

Part (a) of Figure 2 illustrates how this equation models running time on two processors. Because the model uses implicit barriers, we expect that it will yield conservative or pessimistic performance estimates (i.e., predicted execution times possibly greater than actual execution times), as illustrated in part (b) of the figure. We can derive a similar equation for a program that consists of $\text{NSTEPS}$ iterations of $S$:

$$R(T) = (\sum_{i=1}^{\text{NSTEPS}} R(S)) = \text{NSTEPS} \times (\max_{\forall p} R_p(S_1) + \max_{\forall p} R_p(S_2))$$
We demonstrate the use of this simple performance model in Sections 3, 4, and 5.

The *Instantiation* segment of ADAPT is simply the “plugging in” of benchmark numbers into the closed-form equations developed during the *Analysis* segment, either manually or via an automated tool.

The *Performance Model* phase can (but need not) be done concurrently with the *Simulation* phase. With or without the supplementary simulations, ADAPT can guide the programmer in making decisions about data distributions and granularity during the *Refinement* phase (path (f) in Figure 1).
Simulation. The Simulation of a program consists of writing a simulation program based on the actual application program being modeled. This phase supplements the Performance Model phase, in cases in which the programmer would like to consider additional factors before making data partitioning and granularity decisions for large program runs.

We call the methodology of developing the program while using the performance model with a simulation architecture ADEPT, for Application Development using Experimental Performance Tuning [Rif96]. Like the performance model, these simulations can help guide a programmer in making decisions about data distribution and granularity during the Refinement phase (path (g) in Figure 1).

Briefly, setting up a simulation works as follows. Normally, each routine being benchmarked in the Benchmarking phase is timed repeatedly and the results are combined into an average execution time for the routine. If simulation is to be done, however, rather than simply averaging the results the programmer records their distribution. He or she then writes a program analogous to the application program, with the actual application program statements replaced by the generation of estimated running times based on the distributions observed during benchmarking. Rather than computing the results of the application, the simulation computes the expected running time of the application; this can be done quickly on a single processor as many times as the user desires.

Refinement. In the Refinement phase, choices regarding data distribution and granularity are reconsidered based on expected running times as computed in the Performance Model phase. Using ADAPT and ADEPT, the programmer can return to the Application phase (path (h) in Figure 1) to improve the efficiency of the application if necessary. The performance model (and simulation strategy) may suggest a decision between archetypes (Section 5.1), architectures (Section 5.2), or libraries (Section 5.3).

2.2 Experimental Method

We use a small suite of application programs developed using two different archetypes to explore the Performance Model phase of the methodology described in Section 2.1. This section describes our experimental method.

System specifications. Our experiments were conducted using various combinations of two different archetypes (mesh and mesh-spectral), two different architectures (an IBM SP2 at Argonne National Labs using a straight interconnect, and a network of 166 MHz Pentium personal computers connected by 100Mbps Ethernet), and two different languages/libraries (Fortran M and Fortran with MPI).

Fortran M [FC95] consists of a small set of extensions to Fortran for modular parallel programming. In Fortran M, tasks and channels are represented
explicitly, allowing the design of structured, unstructured, and asynchronous communication patterns for task-parallel programs. In addition, Fortran M gives control over the mapping of tasks to processors. We use the TCP/IP-based implementation of Fortran M for our Fortran-M based programs.

MPI [Mes94, SOHL'96], the Message Passing Interface, is a standard, portable message-passing system that defines the syntax and semantics of a package of library routines useful to a wide range of applications written in Fortran or C. Several free, well-tested, efficient implementations of MPI exist, both for distributed memory multicomputers and networks of personal computers and workstations.

**Environment.** Experiments were performed on otherwise unloaded computer nodes, with one application process per node, but in an environment in which communications hardware was also supporting other users. Since execution times were consistent across multiple runs, we assume that this sharing of communications hardware did not greatly affect our results.

**Measurement of execution times.** We measured two kinds of execution times:

- Elapsed “total” time, measured using the Unix `time` command, includes the time required to start the processes.
- Elapsed “process” time, measured by calling the Unix `gettimeofday` system function from within each process, does not include any overhead associated with starting and ending processes.

That is, “total” time is measured from program initiation to program termination, while “process” time is measured from process initiation to process termination. Note that we do not measure elapsed “computational kernel” times, which might show more scalability for our algorithms.

**Averaging.** Measurements are averaged over several trials, with high and low outliers discarded. Every experiment is done at least twice to verify the consistency of the results.

**Presentation of results.** We provide tables containing benchmark measurements, and graphs illustrating observed running times versus those computed using the performance model. In the tables, times are rounded to the nearest integer, so very small times are shown as zero. We plot execution times rather than speedups; each plot shows the following:

- *Ideal* time is sequential execution time (on 1 processor) divided by number of processors — that is, time required for a program with ideal speedup.
• *Actual* time is observed time as measured by our experiments.

• *Expected* time is calculated using the appropriate performance model and the results of our benchmark experiments.

The full text of the experimental parallel programs, sequential programs, and benchmark programs are presented in Appendix A and Appendix B.

3 Mesh Applications

In an application based on the mesh archetype [Mas96], data is based on an \( N \)-dimensional grid (\( N = 1, 2, \) or \( 3 \)), with one or more variables per cell (grid point), and computation consists of some sequence of the following operations:

• Computing, for each cell, new values for one or more variables, based on old values of variables in that cell and nearby cells (for example, neighbors or next-to-neighbors).

• (Optionally) reading in values for a grid variable.

• (Optionally) writing out values for a grid variable.

• (Optionally) computing a global reduction (for example, global maximum or global sum) over the whole grid.

Frequently the compute-new-values and reduction operations are performed repeatedly in a time-step loop. Figure 3 illustrates the basic operation of computing new values in terms of old values, in a two-dimensional grid.

Mesh computations are readily parallelized for distributed-memory architectures using the following approach: The \( N \)-dimensional grid is distributed over an \( N \)-dimensional grid of processes; computation of new values for grid variables is similarly distributed. A separate (optional) host process is used for reads/writes involving a whole array. Non-grid variables (for example, global constants and reduction variables) are duplicated in each process and their values are kept consistent. This approach is discussed by Massingill [Mas96], including details about how to parallelize sequential code and about how to build an application program from the archetype-provided code template and library. The library includes routines that encapsulate the necessary communication operations (host-to-grid and grid-to-host redistribution, boundary exchange, reductions, and broadcast) and a number of utility routines for index manipulation and other housekeeping. This archetype code (template and library) has been implemented in Fortran M, Fortran with Intel’s NX Library, and Fortran with p4 [BL94]. All implementations have essentially the same application programmer interface, so applications developed using one implementation are trivially ported to another.
new_val(i,j) = 
   fold_val(i,j),
   old_val(i-1,j),
   old_val(i+1,j),
   old_val(i,j-1),
   old_val(i,j+1))

Figure 3: Basic mesh computation.

We note that for the two mesh archetype applications given in Sections 3.1 and 3.2, the total process count includes a host process for performing I/O, which affects performance. There is also a no-host-process version of the mesh archetype, but doing I/O with that version is somewhat more complicated, and in the applications described in this paper we chose programming simplicity over performance where such trade-offs had to be made.

3.1 Heat Diffusion

The heat diffusion application [Mas96] solves the one-dimensional heat diffusion equation:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$

using the approximation:

$$\frac{U(x_i, t_{k+1}) - U(x_i, t_k)}{\Delta t} = \frac{U(x_{i+1}, t_k) - 2U(x_i, t_k) + U(x_{i-1}, t_k)}{\Delta x^2}$$

A sequential program for this computation is straightforward. It maintains two copies of variable $U$, one for the current time step ($uk$) and one for the next time step ($ukp1$). At each time step, it computes the values of $ukp1$ based on the values of $uk$. The two boundary points are handled differently; they maintain a constant value.

An equivalent parallel program using the mesh archetype is similar. Grid-based variables $uk$ and $ukp1$ are distributed among grid processes. Each local section is surrounded by a “ghost boundary” of width one, to be used to hold
values from neighboring processes. The whole array is initialized in the host process and then copied to the grid processes. (It could also be initialized directly in the grid processes; this approach was chosen for simplicity.) At each time step, the ghost boundaries are updated (by calling the archetype’s boundary-exchange routine) before they are used in the grid computation. The special handling for the (global) boundary points is provided by using an archetype library routine to determine which points in each local section are in the interior of the global array. The grid values are then copied back to the host process for printing. The code executed by the host and grid processes has the same high-level structure; both execute the time-step loop, for example. This ensures that proper synchronization is maintained. Both programs appear in Appendix A.1.

**Benchmarking.** The computational benchmark measures values for the following times: $T_{\text{overhead}}$ (start and terminate process), $T_{\text{init}}$ (initialize grid values), $T_{\text{comp}}$ (calculate new values for all grid points), and $T_{\text{output}}$ (output results). Results are given in Table 1. Observe that results for this benchmark are independent of the choice of archetype implementation.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{overhead}}$</td>
<td>150</td>
</tr>
<tr>
<td>$T_{\text{init}}$</td>
<td>27</td>
</tr>
<tr>
<td>$T_{\text{comp}}$</td>
<td>124</td>
</tr>
<tr>
<td>$T_{\text{output}}$</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 1: Results of computational benchmark for the mesh heat diffusion application, running on a single node of the IBM SP2 using Fortran. Grid size is 640,000 points. Times are in milliseconds.

The communication benchmark measures values for the following times: $T_{\text{overhead}}$ (start and terminate processes), $T_{\text{HtoG}}$ (redistribute data, host to grid), $T_{\text{intersect}}$ (compute appropriate local indices), $T_{\text{update,bdry}}$ (update ghost boundaries by exchanging data with neighboring processes), and $T_{\text{GtoH}}$ (redistribute data, grid to host). We ran this benchmark on 1, 2, 4, 8, 16, and 32 processors (plus a “host” processor, as described earlier). Results are given in Table 2.

The computational and communication benchmark programs appear in Appendix A.1.

**Performance Model.** We use our performance model and the program, given in Appendix A.1, to compute estimated running time in terms of the preceding
Table 2: Results of communication benchmark for the mesh heat diffusion application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 640,000 points. Times are in milliseconds.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs) on n nodes, not including host node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 1</td>
</tr>
<tr>
<td>$T_{\text{overhead}}$</td>
<td>8650</td>
</tr>
<tr>
<td>$T_{\text{HtoG}}$</td>
<td>556</td>
</tr>
<tr>
<td>$T_{\text{xintersect}}$</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{update_bdry}}$</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{GtoH}}$</td>
<td>530</td>
</tr>
</tbody>
</table>

list of benchmark values and two additional program parameters — NSTEPS (number of loop iterations) and XPROCS (number of grid processes) — as follows. First, a high-level decomposition gives us values for $T_{\text{elapsed}}$ (total estimated elapsed time) and $T_{\text{process}}$ (total estimated process time, excluding process-setup overhead):

$$T_{\text{elapsed}} = T_{\text{process}} + T_{\text{overhead}}$$
$$T_{\text{process}} = T_{\text{startup}} + T_{\text{computation}} + T_{\text{finish}}$$

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application programs as previously described. For the sequential version, the equations are as follows:

$$T_{\text{startup}} = T_{\text{init}}$$
$$T_{\text{computation}} = \text{NSTEPS} \times T_{\text{comp}}$$
$$T_{\text{finish}} = T_{\text{output}}$$

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines, as follows:

$$T_{\text{startup}} = T_{\text{init}} + T_{\text{HtoG}} + T_{\text{xintersect}}$$
$$T_{\text{computation}} = \text{NSTEPS} \times (\frac{T_{\text{comp}}}{XPROCS} + T_{\text{update_bdry}})$$
$$T_{\text{finish}} = T_{\text{GtoH}} + T_{\text{output}}$$
**Experimental Results.** For this experiment we used the following values of the program parameters:

\[
\begin{align*}
\text{nsteps} &= 2000 \\
\text{xprocs} &= n
\end{align*}
\]

where \(n\) is the number of grid (non-“host”) processors (1, 2, 4, 8, 16, or 32). Table 3 and Figures 4 and 5 compare predicted with observed times. For this experiment, predicted times generally agreed well with observed times, with predicted times being, as we expected, somewhat conservative. Our model also correctly predicts the scalability of the application, validating its utility in helping programmers choose granularity.

<table>
<thead>
<tr>
<th></th>
<th>Time (secs) on (n) nodes, not including host node:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(n = 1)</td>
</tr>
<tr>
<td>Expected Elapsed Time</td>
<td>258</td>
</tr>
<tr>
<td>Actual Elapsed Time</td>
<td>237</td>
</tr>
<tr>
<td>Expected Process Time</td>
<td>249</td>
</tr>
<tr>
<td>Actual Process Time</td>
<td>231</td>
</tr>
</tbody>
</table>

Table 3: Execution times for the mesh heat diffusion application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 640,000 grid points and 2000 steps. Times are in seconds. See Figures 4 and 5 for corresponding graphs.

**3.2 Poisson Solver**

This application [Mas96], based on the discussion of the Poisson problem by Van de Velde [VdV94], solve the equation

\[-\frac{\partial^2 U}{\partial x^2} - \frac{\partial^2 U}{\partial y^2} = f(x, y)\]

with Dirichlet boundary condition

\[u(x, y) = g(x, y)\]

using Jacobi iteration; that is, by discretizing the problem domain and applying the following operation to all interior points until convergence is reached:

\[4u_{i,j}^{(k+1)} = h^2 f_{i,j} + u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}\]
Figure 4: Elapsed times for the mesh heat equation application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using FTAM, without the crossbar switch. Problem size is 640,000 grid points and 2000 steps. Times are in seconds. See Table 3 for corresponding table.

Figure 5: Process times for the mesh heat equation application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using FTAM, without the crossbar switch. Problem size is 640,000 grid points and 2000 steps. Times are in seconds. See Table 3 for corresponding table.
Convergence is said to be reached when the maximum of
\[ |u_{(i,j)}^{(k+1)} - u_{(i,j)}^{(k)}| \]
falls below a specified tolerance.

A sequential program for this computation maintains two copies of variable \( u \), one for the current iteration \( u_k \) and one for the next iteration \( u_{k+1} \). At each iteration, it computes the values of \( u_{k+1} \) based on the values of \( u_k \). The boundary points are handled differently; they maintain a constant value. Every \texttt{NCHECK} the maximum difference between \( u_k \) and \( u_{k+1} \) is computed to check for convergence. At each step, values are copied back from \( u_{k+1} \) to \( u_k \) (for the sake of simplicity, at a modest cost in performance).

An equivalent parallel program using the mesh archetype is similar: Grid-based variables \( u_k \) and \( u_{k+1} \) are distributed among grid processes. Each local section is surrounded by a “ghost boundary” of width one, to be used to hold values from neighboring processes. The whole grid is initialized in the host process and then copied to the grid processes. (It could also be initialized directly in the grid processes; this approach was chosen for simplicity.) At each time step, the ghost boundaries are updated (by calling the archetype’s boundary-exchange routine) before they are used in the grid computation. The special handling for the (global) boundary points is provided by using archetype library routines to determine which points in the local section are in the interior of the global array. Computing a global maximum for the convergence test is accomplished by computing a local maximum in each grid process and then calling an archetype library routine to find the global maximum. When convergence is reached (or \texttt{MAXSTEPS} iterations have been performed), grid values are copied back to the host process for printing. The code executed by the host and grid processes has the same high-level structure; both execute the main loop, for example, including the convergence test. This ensures that proper synchronization is maintained. Both programs appear in Appendix A.2.

Note that this problem can also be solved using the mesh-spectral archetype, as described in Section 4.2. We compare the performances of the two implementations (mesh and mesh-spectral) in Section 5.1.

**Benchmarking.** The computational benchmark measures values for the following times: \( T_{\text{overhead}} \) (start and terminate process), \( T_{\text{init}} \) (initialize grid values), \( T_{\text{comp}} \) (calculate new values for all grid points), \( T_{\text{check}} \) (check for convergence), \( T_{\text{copy}} \) (copy new values back to \( u_k \)), and \( T_{\text{output}} \) (output results). Results are given in Table 4. Observe that results for this benchmark are independent of the choice of archetype implementation.

The communication benchmark measures values for the following times: \( T_{\text{overhead}} \) (start and terminate processes), \( T_{\text{global}} \) (index manipulation), \( T_{\text{GtoH}} \) (redistribute data, grid to host), \( T_{\text{HtoG}} \) (redistribute data, host to grid), \( T_{\text{merge}} \) (merge local maxima into global
Table 4: Results of computational benchmark for the mesh Poisson solver application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{\text{overhead}} )</td>
<td>100</td>
</tr>
<tr>
<td>( T_{\text{init}} )</td>
<td>247</td>
</tr>
<tr>
<td>( T_{\text{comp}} )</td>
<td>430</td>
</tr>
<tr>
<td>( T_{\text{check-converge}} )</td>
<td>128</td>
</tr>
<tr>
<td>( T_{\text{copy-values}} )</td>
<td>45</td>
</tr>
<tr>
<td>( T_{\text{output}} )</td>
<td>15</td>
</tr>
</tbody>
</table>

maximum), \( T_{\text{update-ldry}} \) (update ghost boundaries by exchanging data with neighboring processes), \( T_{\text{interpct}} \) (index manipulation), and \( T_{\text{yintercept}} \) (index manipulation). We ran this benchmark on 1, 4, 9, 16, 25, and 36 processors (plus a “host” processor, as described earlier). Results are given in Table 5.

Table 5: Results of communication benchmark for the mesh Poisson solver application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs) on ( n ) nodes, not including host node</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 1 )</td>
<td>( n = 4 )</td>
</tr>
<tr>
<td>( T_{\text{overhead}} )</td>
<td>6</td>
</tr>
<tr>
<td>( T_{\text{iglobal}} )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{\text{jglobal}} )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{\text{GtoH}} )</td>
<td>618</td>
</tr>
<tr>
<td>( T_{\text{HtoG}} )</td>
<td>665</td>
</tr>
<tr>
<td>( T_{\text{merge-real-maxabs}} )</td>
<td>7</td>
</tr>
<tr>
<td>( T_{\text{update-ldry}} )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{\text{interpct}} )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{\text{yinterpct}} )</td>
<td>0</td>
</tr>
</tbody>
</table>

The computational and communication benchmark programs appear in Ap-
Performance Model. We use our performance model and the program, given in Appendix A.2, to compute estimated running time in terms of the preceding list of benchmark values and a few additional program parameters — NSTEPS (number of loop iterations), NCHECK (frequency of convergence checking), and XPROCS and YPROCS (dimensions of process grid, implying a total of XPROCS × YPROCS grid processes) — as follows First, a high-level decomposition gives us values for T\textsubscript{elapsed} (total estimated elapsed time) and T\textsubscript{process} (total estimated process time, excluding process-setup overhead):

\[
T_{\text{elapsed}} = T_{\text{process}} + T_{\text{overhead}}
\]
\[
T_{\text{process}} = T_{\text{startup}} + T_{\text{computation}} + T_{\text{check}} + T_{\text{copy}} + T_{\text{finish}}
\]

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application programs as previously described. For the sequential version, the equations are as follows:

\[
\begin{align*}
T_{\text{startup}} &= T_{\text{init}} \\
T_{\text{computation}} &= \text{NSTEPS} \times T_{\text{comp}} \\
T_{\text{check}} &= \frac{\text{NSTEPS}}{\text{NCHECK}} \times T_{\text{check,converge}} \\
T_{\text{copy}} &= \text{NSTEPS} \times T_{\text{copy,values}} \\
T_{\text{finish}} &= T_{\text{output}}
\end{align*}
\]

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines, as follows:

\[
\begin{align*}
T_{\text{startup}} &= T_{\text{init}} + T_{\text{HtoG}} + T_{\text{XtoG}} + T_{\text{Xint,intersect}} + T_{\text{Yint,intersect}} + T_{\text{iglobal}} + T_{\text{jglobal}} \\
T_{\text{computation}} &= \text{NSTEPS} \times \left( \frac{T_{\text{comp}}}{\text{XPROCS} \times \text{YPROCS}} + T_{\text{update,\_dry}} \right) \\
T_{\text{check}} &= \frac{\text{NSTEPS}}{\text{NCHECK}} \times \left( \frac{T_{\text{check,converge}}}{\text{XPROCS} \times \text{YPROCS}} + T_{\text{merge,\_real,\_maxabs}} \right) \\
T_{\text{copy}} &= \text{NSTEPS} \times \left( \frac{T_{\text{copy,values}}}{\text{XPROCS} \times \text{YPROCS}} \right) \\
T_{\text{finish}} &= T_{\text{GtoH}} + T_{\text{output}}
\end{align*}
\]
Experimental Results. For this experiment we used the following values of the program parameters:

\[
\begin{align*}
\text{NSTEPS} &= 1000 \\
\text{NCHECK} &= 10 \\
\text{XPROCS} &= \text{YPROCS} = \sqrt{n}
\end{align*}
\]

where \( n \) is the number of grid (non-“host”) processors (1, 4, 9, 16, 25, or 36). Table 6 and Figures 6 and 7 compare predicted with observed times. For this experiment, predicted times generally agreed fairly well with observed times, with predicted times being, as we expected, somewhat conservative. Our model did less well for this application than for the heat equation application in predicting scalability, but it still came fairly close.

<table>
<thead>
<tr>
<th></th>
<th>Time (secs) on ( n ) nodes, not including host node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( n = 1 )</td>
</tr>
<tr>
<td>Expected Elapsed Time</td>
<td>496</td>
</tr>
<tr>
<td>Actual Elapsed Time</td>
<td>522</td>
</tr>
<tr>
<td>Expected Process Time</td>
<td>490</td>
</tr>
<tr>
<td>Actual Process Time</td>
<td>517</td>
</tr>
</tbody>
</table>

Table 6: Execution times for the mesh Poisson solver application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figures 6 and 7 for corresponding graphs.

4 Mesh-Spectral Applications

In an application written using the mesh archetype [DM96]: Data is based on three-dimensional grids (arrays), with one- and two-dimensional grids considered as special cases of three-dimensional grids; a computation may contain multiple grids of different dimensions. Computation consists of some sequence of the following operations:

- Neighbor operations in which new values are computed for each point in a grid based on values at that point and nearby points.
- Row operations, in which new values are computed for each point in a grid based on values in the same row.
Figure 6: Elapsed times for the mesh Poisson solver application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 6 for corresponding table.

Figure 7: Process times for the mesh Poisson solver application, running on the specified number of nodes (plus a “host” node) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 6 for corresponding table.
- Column operations, defined analogously.
- Reduction operations over a grid (for example, global maximum).

Figures 8 and 9 illustrate two of these operations (a neighbor operation and a row operation respectively) in a two-dimensional grid.

\[
\text{new}_{-}\text{val}(i,j) = \text{f(old}_{-}\text{val}(i,j), \\
\quad \text{old}_{-}\text{val}(i-1,j), \\
\quad \text{old}_{-}\text{val}(i+1,j), \\
\quad \text{old}_{-}\text{val}(i,j-1), \\
\quad \text{old}_{-}\text{val}(i,j+1))}
\]

Figure 8: Neighbor operation.

\[
\text{new}_{-}\text{val}(i,j) = \text{f(old}_{-}\text{val}(i,1), \\
\quad \text{old}_{-}\text{val}(i,2), \\
\quad \text{.....} \\
\quad \text{old}_{-}\text{val}(i,N))}
\]

Figure 9: Row operation.

Mesh-spectral computations are readily parallelized using the following approach: The overall structure of the computation is based on the SPMD (Single
Program, Multiple Data) model; that is, it consists of some number $N$ of processes all executing the same program, each on its own data. Each 3-dimensional data grid is distributed over the processes based on a 3-dimensional process grid of some or all of the $N$ processes; computation of new values for grid variables is similarly distributed. In the course of a computation, a data grid can be redistributed (that is, the process grid over which it is distributed can be changed); this is usually done when one distribution is convenient for part of the computation and a different distribution is convenient for another part of the computation. Non-grid variables (for example, global constants and reduction variables) are duplicated in each process; their values are kept consistent. This approach is discussed by Massingill and Davis [DM96], including details about how to parallelize sequential code and about how to build an application program from the archetype-provided code template and library. The library includes routines that encapsulate the necessary communication operations (redistribution, boundary exchange, reductions, and broadcast) and a number of utility routines for index manipulation and other housekeeping. This archetype code (template and library) has been implemented in Fortran M and Fortran with MPI. All implementations have essentially the same application programmer interface, so applications developed using one implementation are trivially ported to another.

4.1 Two-dimensional Fast Fourier Transform

This application [DM96] performs a two-dimensional FFT in place. As described in Numerical Recipes [PFTV86], computing a two-dimensional FFT in place is accomplished by performing a one-dimensional FFT on each row of a two-dimensional array, and then performing a one-dimensional FFT on each column of the resulting two-dimensional array. Optimality of the 2D FFT depends largely on the choice of algorithm (and implementation) for the 1D FFT.

In our application, the actual FFTs are performed using a sequential subroutine library that allows the computation of FFTs on a set of vectors with a single subroutine call. The sequential version of the application is straightforward (first use the library subroutine to perform 1D FFTs on each row, and then use it to perform 1D FFTs on each column. A parallel version based on the mesh-spectral archetype is not much more complicated: In order to perform the FFT computations using the same sequential subroutine library, the program employs two distributions for the two-dimensional array on which the FFT is to be performed: a distribution by rows (which, for each row, puts all data in a single process, allowing it to be processed with a sequential FFT subroutine), and a distribution by columns (which has the same effect on columns). Data is initially distributed by rows; after the FFT-by-rows is performed, it is redistributed by columns and the FFT-by-columns is performed. It is then redistributed by rows before being written out. The FFT computation (calculation by rows, redistribution, calculation by columns, and then redistribution
again) is repeated several times to reduce the proportion of total application time spent on I/O.

Our 1D FFT code was written by Clive Temperton for the Meteorological Office in England. The code was originally designed for vector machines like the Cray; today, the library is widely available at all supercomputer centers. Note that the details of the one-dimensional FFT implementation are not relevant to the parallelization, and while they affect performance, we use the same one-dimensional FFT in both the sequential and parallel programs. Thus, overall performance could be improved by choosing a faster 1D FFT, without changing the parallel aspects of the code.

We note that [PFTVS6] is not the most efficient implementation of an FFT, but it is well-understood and easily implemented using the mesh-spectral archetype; in this paper, we use this algorithm primarily to illustrate the performance model, rather than attempt to achieve the fastest possible FFT (as is the focus of other work, such as [Win78]). Duhamel and Vetterli provide an excellent survey of FFTs [DV90]. A good comparison of the FFT algorithm we use with more efficient ones (such as a split-radix algorithm) on a vanilla workstation is given in [Arn96] (although, with a multicomputer, a simpler butterfly structure might be better for more actual computation [Hat96]).

**Benchmarking**. The computational benchmark measures values for the following times: $T_{\text{overhead}}$ (start and terminate process), $T_{\text{read}fft}$ (set up grid and read input data), $T_{\text{init}}$ (initialize for FFT), $T_{\text{row}fft}$ (perform FFTs on rows), $T_{\text{col}fft}$ (perform FFTs on columns), and $T_{\text{write}fft}$ (write output data). Results are given in Table 7. Observe that results for this benchmark are independent of the choice of archetype implementation.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{overhead}}$</td>
<td>2800</td>
</tr>
<tr>
<td>$T_{\text{read}fft}$</td>
<td>16161</td>
</tr>
<tr>
<td>$T_{\text{init}}$</td>
<td>3</td>
</tr>
<tr>
<td>$T_{\text{row}fft}$</td>
<td>2854</td>
</tr>
<tr>
<td>$T_{\text{col}fft}$</td>
<td>3335</td>
</tr>
<tr>
<td>$T_{\text{write}fft}$</td>
<td>12934</td>
</tr>
</tbody>
</table>

Table 7: Results of computational benchmark for the mesh-spectral 2D FFT application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

The communication benchmark measures values for the following times:
\( T_{\text{overhead}} \) (start and terminate processes), \( T_{\text{col} \rightarrow \text{row}} \) (redistribute data, columns to rows), \( T_{\text{row} \rightarrow \text{col}} \) (redistribute data, rows to columns), \( T_{\text{data widths}} \) (housekeeping), \( T_{\text{local pos}} \) (housekeeping), \( T_{\text{set mesh}} \) (set up data grid), \( T_{\text{read mesh}} \) (perform communication associated with reading data), and \( T_{\text{write mesh}} \) (perform communication associated with writing data). We ran this benchmark on 1, 2, 4, 8, 16, and 32 processors. Results are given in Table 8.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs) on n nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 1</td>
</tr>
<tr>
<td>( T_{\text{overhead}} )</td>
<td>30930</td>
</tr>
<tr>
<td>( T_{\text{col} \rightarrow \text{row}} )</td>
<td>3197</td>
</tr>
<tr>
<td>( T_{\text{row} \rightarrow \text{col}} )</td>
<td>3195</td>
</tr>
<tr>
<td>( T_{\text{data widths}} )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{\text{local pos}} )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{\text{set mesh}} )</td>
<td>0</td>
</tr>
<tr>
<td>( T_{\text{read mesh}} )</td>
<td>619</td>
</tr>
<tr>
<td>( T_{\text{write mesh}} )</td>
<td>634</td>
</tr>
</tbody>
</table>

Table 8: Results of communication benchmark for the mesh-spectral 2D FFT application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

The computational and communication benchmark programs appear in Appendix B.1.

**Performance Model.** We use our performance model and the program, given in Appendix B.1, to compute estimated running time in terms of the preceding list of benchmark values and two additional program parameters — \( \text{NREPEATS} \) (number of times to repeat the FFT) and \( \text{NPROCS} \) (number of processes) — as follows. First, a high-level decomposition gives us values for \( T_{\text{elapsed}} \) (total estimated elapsed time) and \( T_{\text{process}} \) (total estimated process time, excluding process-setup overhead):

\[
T_{\text{elapsed}} = T_{\text{process}} + T_{\text{overhead}}
\]

\[
T_{\text{process}} = \text{NREPEATS} \times (T_{\text{startup}} + T_{\text{computation}} + T_{\text{finish}})
\]

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application
programs as previously described. For the sequential version, the equations are as follows:

\[ T_{\text{startup}} = T_{\text{read}} + T_{\text{init}} \]
\[ T_{\text{computation}} = T_{\text{row}} + T_{\text{col}} \]
\[ T_{\text{finish}} = T_{\text{write}} \]

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines, as follows:

\[ T_{\text{startup}} = T_{\text{set mesh}} + T_{\text{read}} + T_{\text{read mesh}} + T_{\text{init}} \]
\[ T_{\text{computation}} = T_{\text{local}} + T_{\text{data widths}} + \left( T_{\text{row}} \frac{T_{\text{col}}}{\text{NPROCS}} \right) + T_{\text{full col}} + \]
\[ T_{\text{local}} + T_{\text{data widths}} + \left( T_{\text{row}} \frac{T_{\text{col}}}{\text{NPROCS}} \right) + T_{\text{col to row}} \]
\[ T_{\text{finish}} = T_{\text{write}} + T_{\text{write mesh}} \]

**Experimental Results.** For this experiment we used the following values of the program parameters:

\[ \text{NREPEATS} = 10 \]
\[ \text{NPROCS} = n \]

where \( n \) is the number of processors (1, 2, 4, 8, 16, or 32). Table 9 and Figures 10 and 11 compare predicted with observed times. For this experiment, predicted times generally agreed well with observed times, with predicted times mostly being, as we expected, somewhat pessimistic. (The exception is the predictions for 4 nodes, which in both cases were optimistic.)

The overall performance of this application is admittedly disappointing. In part this is because the application reads and writes the whole array; including this substantial I/O degrades performance but makes the program slightly more realistic and demonstrates that the performance model is compatible with the archetype's I/O handling. However, it appears that this application simply does not scale very well; for more than a few processors, performance gains obtained by distributing the computation are largely negated by the additional time required for interprocess communication. Possibly this could be overcome by optimizing the archetype library (the current implementation is an unoptimized proof-of-concept version, which could be replaced by an optimized version, thus improving performance of all mesh-spectral applications).

Despite the application's unimpressive performance, however, this experiment validates our model, since predicted execution times were close to actual
execution times. A situation such as this one suggests an additional use for a good performance model — deciding on the basis of the model that a particular parallelization scheme is not likely to be effective without actually coding it up and trying it.

<table>
<thead>
<tr>
<th></th>
<th>Time (secs) on n nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n = 1$  $n = 2$  $n = 4$  $n = 8$  $n = 16$  $n = 32$</td>
</tr>
<tr>
<td>Expected Elapsed Time</td>
<td>160     101     69     69     68     78</td>
</tr>
<tr>
<td>Actual Elapsed Time</td>
<td>157     100     80     51     49     60</td>
</tr>
<tr>
<td>Expected Process Time</td>
<td>156     96      63     60     51     45</td>
</tr>
<tr>
<td>Actual Process Time</td>
<td>153     95      73     41     32     29</td>
</tr>
</tbody>
</table>

Table 9: Execution times for the mesh-spectral 2D FFT application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Problem size is 800 by 800 points and 10 repetitions. Times are in seconds. See Figures 10 and 11 for corresponding graphs.

Figure 10: Elapsed times for the mesh-spectral 2D FFT application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Problem size is 800 by 800 points and 10 repetitions. See Table 9 for corresponding table.
Figure 11: Process times for the mesh-spectral 2D FFT application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Problem size is 800 by 800 points and 10 repetitions. See Table 9 for corresponding table.

4.2 Poisson Solver

This application is another implementation, using the mesh-spectral archetype this time, of the Poisson solver described in Section 3.2, with one additional feature: Values for stepsize (H) and convergence tolerance (TOL) are to be read at runtime from standard input or an input file. Both sequential and parallel versions are very similar to those described in Section 3.2, except that the parallel version uses the mesh-spectral archetype library rather than the mesh archetype library, and both versions read in stepsize and tolerance. (The parallel version performs this read in the archetype’s designated I/O process and then uses broadcast to copy their values to the other processes.) We compare the performance of the mesh-spectral implementation to that of the mesh implementation in Section 5.1.

Benchmarking. The computational benchmark measures values for the following times: $T_{\text{overhead}}$ (start and terminate process), $T_{\text{read, const}}$ (read constants), $T_{\text{init}}$ (initialize grid values), $T_{\text{comp}}$ (calculate new values for all grid points), $T_{\text{check, converge}}$ (check for convergence), $T_{\text{copy values}}$ (copy new values back to uk), and $T_{\text{output}}$ (output results). Results are given in Table 10. Observe that results for this benchmark are independent of the choice of archetype implementation.

The communication benchmark measures values for the following times:
<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{overhead}}$</td>
<td>100</td>
</tr>
<tr>
<td>$T_{\text{read const}}$</td>
<td>15</td>
</tr>
<tr>
<td>$T_{\text{init}}$</td>
<td>222</td>
</tr>
<tr>
<td>$T_{\text{comp}}$</td>
<td>427</td>
</tr>
<tr>
<td>$T_{\text{check converge}}$</td>
<td>134</td>
</tr>
<tr>
<td>$T_{\text{copy values}}$</td>
<td>62</td>
</tr>
<tr>
<td>$T_{\text{output}}$</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 10: Results of computational benchmark for the mesh-spectral Poisson solver application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

The computational and communication benchmark programs appear in Appendix B.2.

**Performance Model.** We use our performance model and the program, given in Appendix B.2, to compute estimated running time in terms of the preceding list of benchmark values and a few additional program parameters — $\text{NSTEPS}$ (number of loop iterations), $\text{NCHECK}$ (frequency of convergence checking), and $\text{NXPROCS}$ and $\text{NYPROCS}$ (dimensions of process grid, implying a total of $\text{NXPROCS} \times \text{NYPROCS}$ processes) — as follows. First, a high-level decomposition gives us values for $T_{\text{elapsed}}$ (total estimated elapsed time) and $T_{\text{process}}$ (total estimated process time, excluding process-setup overhead):

$$T_{\text{elapsed}} = T_{\text{process}} + T_{\text{overhead}}$$
$$T_{\text{process}} = T_{\text{startup}} + T_{\text{computation}} + T_{\text{check}} + T_{\text{copy}} + T_{\text{finish}}$$

We can then write down equations for each term on the right-hand side of the above equations based on applying our performance model to the application programs as previously described. For the sequential version, the equations are
Table 11: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Measurement} & n = 1 & n = 4 & n = 9 & n = 16 & n = 25 & n = 36 \\
\hline
\mathcal{T}_{\text{overhead}} & 3850 & 9630 & 15950 & 27380 & 48330 & 86150 \\
\mathcal{T}_{\text{setup mesh}} & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathcal{T}_{\text{blk to one}} & 2928 & 1957 & 1365 & 1219 & 954 & 1241 \\
\mathcal{T}_{\text{one to blk}} & 2951 & 2474 & 2051 & 2026 & 2222 & 1792 \\
\mathcal{T}_{\text{bdry exchg}} & 0 & 11 & 13 & 14 & 16 & 20 \\
\mathcal{T}_{\text{bcast}} & 0 & 4 & 14 & 35 & 75 & 145 \\
\mathcal{T}_{\text{data bounds}} & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathcal{T}_{\text{global max dp}} & 0 & 12 & 25 & 53 & 96 & 171 \\
\mathcal{T}_{\text{intersect}} & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathcal{T}_{\text{local pos}} & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathcal{T}_{\text{local to global}} & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathcal{T}_{\text{pack}} & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathcal{T}_{\text{unpack}} & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

as follows:

\[
\mathcal{T}_{\text{startup}} = \mathcal{T}_{\text{read const}} + \mathcal{T}_{\text{init}} \\
\mathcal{T}_{\text{computation}} = \text{NSTEPS} \times \mathcal{T}_{\text{comp}} \\
\mathcal{T}_{\text{check}} = \frac{\text{NSTEPS}}{\text{NCHECK}} \times \mathcal{T}_{\text{check converge}} \\
\mathcal{T}_{\text{copy}} = \text{NSTEPS} \times \mathcal{T}_{\text{copy values}} \\
\mathcal{T}_{\text{finish}} = \mathcal{T}_{\text{output}}
\]

For the parallel version, the equations reflect the division of computation among processes and also the inclusion of communication and housekeeping routines, as follows:

\[
\mathcal{T}_{\text{startup}} = \mathcal{T}_{\text{local pos}} + \mathcal{T}_{\text{read const}} + \mathcal{T}_{\text{bcast}} + \mathcal{T}_{\text{setup mesh}} + \mathcal{T}_{\text{local pos}} + \\
\mathcal{T}_{\text{init}} + \mathcal{T}_{\text{one to blk}} + \mathcal{T}_{\text{local pos}}
\]
\[ T_{\text{computation}} = (T_{\text{pack}} \times 3) + (T_{\text{unpack}} \times 3) + \\
T_{\text{data bounds}} + T_{\text{intersect}} + T_{\text{local to global}} + \\
\text{NSTEPS} \times \left( \frac{T_{\text{comp}}}{XPROCS \times YPROCS} + T_{\text{bdry excchg}} \right) \]

\[ T_{\text{check}} = \frac{\text{NSTEPS}}{\text{NCHECK}} \times \left( \frac{T_{\text{check converge}}}{XPROCS \times YPROCS} + T_{\text{global max dp}} \right) \]

\[ T_{\text{copy}} = \text{NSTEPS} \times \left( \frac{T_{\text{copy values}}}{NXPROCS \times NYPROCS} \right) \]

\[ T_{\text{finish}} = T_{\text{output}} \]

**Experimental Results.** For this experiment we used the following values of the program parameters:

\[
\begin{align*}
\text{NSTEPS} &= 1000 \\
\text{NCHECK} &= 10 \\
\text{NXPROCS} &= \text{NYPROCS} = \sqrt{n}
\end{align*}
\]

where \( n \) is the number of processors \((1, 4, 9, 16, 25, \text{or } 36)\). Table 12 and Figures 12 and 13 compare predicted with observed times. For this experiment, predicted times agreed well with observed times. Surprisingly, several predictions were slightly optimistic, but overall the agreement was quite good for this application. Our model also correctly predicts the scalability of the application, validating its utility in helping programmers choose granularity.

<table>
<thead>
<tr>
<th></th>
<th>Time (secs) on ( n ) nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( n = 1 )</td>
</tr>
<tr>
<td>Expected Elapsed Time</td>
<td>512</td>
</tr>
<tr>
<td>Actual Elapsed Time</td>
<td>521</td>
</tr>
<tr>
<td>Expected Process Time</td>
<td>508</td>
</tr>
<tr>
<td>Actual Process Time</td>
<td>516</td>
</tr>
</tbody>
</table>

Table 12: Execution times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds. See Figures 12 and 13 for corresponding graphs.
Figure 12: Elapsed times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds. See Table 12 for corresponding table.

Figure 13: Process times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds. See Table 12 for corresponding table.
5 Evaluating the Performance Model

In this section, we evaluate the utility of our performance model. Through selected experiments, we show that our performance analysis works well when applied to different archetypes (Section 5.1), to different architectures (Section 5.2), and to different communication libraries (Section 5.3).

In addition, we show that our performance model can be used to choose between different data distributions (Section 5.4), and that our performance model can be used to simulate actual program executions to predict expected running times (Section 5.5).

5.1 Performance Analysis Across Archetypes

In this section, we demonstrate that our performance analysis works well when used to compare running times for the same application developed using different archetypes. For this experiment, we employ two versions of the Poisson solver application (described in Sections 3.2 and 4.2), both implemented in Fortran M and running on an IBM SP2, but one using the mesh archetype and one the mesh-spectral archetype.

Benchmarking. The computational and communication benchmark programs for this application appear in Appendix A.2 and Appendix B.2 for the mesh and mesh-spectral versions, respectively. Results of the computational and communication benchmarks for the mesh version of the application, executed using the target archetype implementation and architecture, appear in Tables 4 and 5, respectively. Results of the computational and communication benchmarks for the mesh-spectral version of the application, executed using the target archetype implementation and architecture, appear in Tables 10 and 11, respectively.

Performance Models. Our performance model for the mesh Poisson solver application is given in Section 3.2, based on the program in Appendix A.2. Our performance model for the mesh-spectral Poisson solver application is given in Section 4.2, based on the program in Appendix B.2.

Experimental Results. For this experiment we used the following values of the program parameters:

\[
\text{NSTEPS} = 1000 \\
\text{NCHECK} = 10
\]

For the mesh version, we used

\[
\text{XPROCS} = \text{YPROCS} = \sqrt{n}
\]
where \( n \) is the number of grid (non-“host”) processors (1, 4, 9, 16, 25, or 36). For the mesh-spectral version, we used

\[
\text{nxprocs} = \text{nyprocs} = \sqrt{n}
\]

where \( n \) is the number of processors (1, 4, 9, 16, 25, or 36). Table 13 and Figure 14 compare elapsed times (predicted and observed) for the two programs. Table 14 and Figure 15 compare process times (predicted and observed) for the two programs. These results have been discussed previously (in Sections 3.2 and 4.2); note again that for both versions of the application the model’s predictions about execution times and scaling are generally good. The model also correctly predicts that overall the mesh-spectral version of the application performs better.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Time (secs) on ( n ) nodes, not including host node</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 1 )</td>
<td>( n = 4 )</td>
</tr>
<tr>
<td>Mesh-Spectral Expected Elapsed Time</td>
<td>512</td>
</tr>
<tr>
<td>Mesh-Spectral Actual Elapsed Time</td>
<td>521</td>
</tr>
<tr>
<td>Mesh Expected Elapsed Time</td>
<td>496</td>
</tr>
<tr>
<td>Mesh Actual Elapsed Time</td>
<td>522</td>
</tr>
</tbody>
</table>

Table 13: Elapsed times for the mesh and mesh-spectral Poisson solver applications, running on the specified number of nodes (plus a “host” node for the mesh version) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 14 for corresponding graph.

### 5.2 Performance Analysis Across Architectures

In this section, we demonstrate that our performance analysis works well when used to compare running times for the same application developed using the same archetype, executing on different target machines (with portability as an automatic consequence of using a portable archetype implementation). For this experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), implemented in Fortran with MPI, running on an IBM SP2 and...
Figure 14: Elapsed times for the mesh and mesh-spectral Poisson solver applications, running on the specified number of nodes (plus a “host” node for the mesh version) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 13 for corresponding table.

<table>
<thead>
<tr>
<th></th>
<th>Mesh-Spectral</th>
<th>Mesh-Spectral</th>
<th>Mesh</th>
<th>Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 1$</td>
<td>508</td>
<td>516</td>
<td>490</td>
<td>517</td>
</tr>
<tr>
<td>$n = 4$</td>
<td>142</td>
<td>143</td>
<td>141</td>
<td>140</td>
</tr>
<tr>
<td>$n = 9$</td>
<td>75</td>
<td>75</td>
<td>80</td>
<td>73</td>
</tr>
<tr>
<td>$n = 16$</td>
<td>54</td>
<td>54</td>
<td>59</td>
<td>47</td>
</tr>
<tr>
<td>$n = 25$</td>
<td>49</td>
<td>55</td>
<td>59</td>
<td>34</td>
</tr>
<tr>
<td>$n = 36$</td>
<td>55</td>
<td>63</td>
<td>54</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 14: Process times for the mesh and mesh-spectral Poisson solver applications, running on the specified number of nodes (plus a “host” node for the mesh version) on the IBM SP2 using Fortran M, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 15 for corresponding graph.
on a network of 166 MHz Pentium personal computers connected by 100 Mbps Ethernet.

**Benchmarking.** The computational and communication benchmark programs for this application appear in Appendix B.2. Applying our performance analysis for a particular implementation and architecture requires results from executing both benchmarks on an appropriate system. For the SP2, we can reuse the computational benchmark results shown in Table 10, since the target architecture is the same. We must, however, rerun the communication benchmark using the MPI-based archetype implementation. As before, we ran this benchmark on 1, 4, 9, 16, 25, and 36 processors; results are given in Table 15. For the network of Pentiums, we must rerun both computational and communication benchmarks. Results of running the computational benchmark on one Pentium processor appear in Table 16. Due to a bug in the MPI implementation installed on our target network, we were unable to make use of more than 9 processors, so we ran the communication benchmark for 1, 2, 4, 6, 8, and 9 processors. Results are given in Table 17.

**Performance Model.** Our performance model for this application is given in Section 4.2, based on the program in Appendix B.2, and is applicable to both
Table 15: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran with MPI, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs) on n nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{\text{overhead}})</td>
<td>(n = 1)</td>
</tr>
<tr>
<td>(T_{\text{set_mesh}})</td>
<td>3930</td>
</tr>
<tr>
<td>(T_{\text{blk_to_one}})</td>
<td>3218</td>
</tr>
<tr>
<td>(T_{\text{one_to_blk}})</td>
<td>3243</td>
</tr>
<tr>
<td>(T_{\text{bdry_exchg}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{bcast}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{data_bounds}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{global_max_dp}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{intersect}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{local_pos}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{local_to_global}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{pack}})</td>
<td>0</td>
</tr>
<tr>
<td>(T_{\text{unpack}})</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 16: Results of computational benchmark for the mesh-spectral Poisson solver application, running on a single 166 MHz Pentium using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{\text{overhead}})</td>
<td>150</td>
</tr>
<tr>
<td>(T_{\text{read_const}})</td>
<td>5</td>
</tr>
<tr>
<td>(T_{\text{init}})</td>
<td>271</td>
</tr>
<tr>
<td>(T_{\text{comp}})</td>
<td>941</td>
</tr>
<tr>
<td>(T_{\text{check_converge}})</td>
<td>265</td>
</tr>
<tr>
<td>(T_{\text{copy_values}})</td>
<td>244</td>
</tr>
<tr>
<td>(T_{\text{output}})</td>
<td>40</td>
</tr>
</tbody>
</table>
Table 17: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on a network of 166 MHz Pentiums using Fortran with MPI, communicating over 100 Mbps Ethernet. Grid size is 800 by 800 points. Times are in milliseconds.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>$n = 1$</th>
<th>$n = 2$</th>
<th>$n = 4$</th>
<th>$n = 6$</th>
<th>$n = 8$</th>
<th>$n = 9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{overhead}}$</td>
<td>3000</td>
<td>7530</td>
<td>9930</td>
<td>14830</td>
<td>18000</td>
<td>18150</td>
</tr>
<tr>
<td>$T_{\text{set_mesh}}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{blk_to_one}}$</td>
<td>3957</td>
<td>3805</td>
<td>3307</td>
<td>2310</td>
<td>2243</td>
<td>2133</td>
</tr>
<tr>
<td>$T_{\text{one_to_blk}}$</td>
<td>4115</td>
<td>3738</td>
<td>463</td>
<td>4961</td>
<td>5230</td>
<td>534</td>
</tr>
<tr>
<td>$T_{\text{bdry_exchg}}$</td>
<td>0</td>
<td>11</td>
<td>15</td>
<td>19</td>
<td>209</td>
<td>296</td>
</tr>
<tr>
<td>$T_{\text{cast}}$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$T_{\text{data_bounds}}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{global_max_fip}}$</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>$T_{\text{intersect}}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{local_pos}}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{local_to_global}}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{pack}}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{\text{unpack}}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Experimental Results. For this experiment we used the following values of the program parameters:

$$\text{NSTEPS} = 1000$$
$$\text{NCHECK} = 10$$

On the SP2, we used

$$\text{NXPROCS} = \text{NYPROCS} = \sqrt{n}$$

where $n$ is the number of processors (1, 4, 9, 16, 25, or 36). On the network of Pentiums, we used the following values of $(\text{NXPROCS}, \text{NYPROCS})$: (1,1), (1,2), (2,2), (2,3), (2,4), and (3,3), corresponding to 1, 2, 4, 8, and 9 processors. Table 18 and Figure 16 compare elapsed times (predicted and observed) for the two architectures (network of Pentiums and IBM SP2). Table 19 and Figure 17 compare process times (predicted and observed) for the two architectures. For this experiment, predicted times agreed well with observed times for both architectures. Surprisingly, many predicted times for the network of Pentiums
were optimistic, though not extremely so. For both architectures, our model predicts the scalability of the application pretty well, and it correctly predicts the expected performance difference between the two architectures.

<table>
<thead>
<tr>
<th>Time (secs) on n nodes</th>
<th>SP2 Expected Elapsed Time</th>
<th>SP2 Actual Elapsed Time</th>
<th>Pentium Expected Elapsed Time</th>
<th>Pentium Actual Elapsed Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 1</td>
<td>513</td>
<td>520</td>
<td>1222</td>
<td>1308</td>
</tr>
<tr>
<td>n = 2</td>
<td>-</td>
<td>-</td>
<td>632</td>
<td>712</td>
</tr>
<tr>
<td>n = 4</td>
<td>-</td>
<td>-</td>
<td>337</td>
<td>362</td>
</tr>
<tr>
<td>n = 6</td>
<td>-</td>
<td>-</td>
<td>243</td>
<td>288</td>
</tr>
<tr>
<td>n = 8</td>
<td>-</td>
<td>-</td>
<td>387</td>
<td>342</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time (secs) on n nodes</th>
<th>SP2 Expected Elapsed Time</th>
<th>SP2 Actual Elapsed Time</th>
<th>Pentium Expected Elapsed Time</th>
<th>Pentium Actual Elapsed Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 9</td>
<td>74</td>
<td>75</td>
<td>457</td>
<td>379</td>
</tr>
<tr>
<td>n = 16</td>
<td>56</td>
<td>52</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n = 25</td>
<td>54</td>
<td>52</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>n = 36</td>
<td>61</td>
<td>56</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 18: Elapsed times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 16 for corresponding graph.

5.3 Performance Analysis Across Libraries

In this section, we demonstrate that our performance analysis works well when used to compare running times for the same application developed using the same archetype for the same target machine, using different communication libraries (i.e., different archetype implementations). For this experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), comparing a Fortran M implementation running on an IBM SP2 with an MPI implementation also running on an IBM SP2.

**Benchmarking.** The computational and communication benchmark programs appear in Appendix B.2. As in Section 5.2, applying our performance
Figure 16: Elapsed times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 18 for corresponding table.

analysis for a particular implementation requires results from executing both benchmarks on appropriate system. Since here we are comparing different archetype implementations on the same target architecture, we can use the same computational benchmark results for both, namely the ones presented in Table 10. For the communication benchmark, we need results from executing versions based on both the Fortran M and MPI archetype implementations, presented respectively in Tables 11 and 15.

**Performance Model.** Our performance model for this application is given in Section 4.2, based on the program in Appendix B.2, and is applicable to both implementations.

**Experimental Results.** For this experiment we used the following values of the program parameters:

\[
\begin{align*}
\text{nsteps} &= 1000 \\
\text{nccheck} &= 10 \\
\text{nxprocs} &= \text{nyprocs} = \sqrt{n}
\end{align*}
\]
Table 19: Process times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 17 for corresponding graph.

where $n$ is the number of processors (1, 4, 9, 16, 25, or 36). Table 20 and Figure 18 compare elapsed times (predicted and observed) for the two implementations. Table 21 and Figure 19 compare process times (predicted and observed) for the two implementations. These results have been discussed previously (in Section 4.2 for the Fortran M implementation and Section 5.2 for the MPI implementation); note again that for both implementations the model’s predictions about execution times and scaling are generally good. The model also correctly predicts that the MPI implementation performs better than the Fortran M implementation, suggesting that it could help programmers decide between archetype implementations without trying both.

5.4 Performance Analysis and Data Distributions

In this section, we show that our performance model can be used to predict how performance is affected by data distribution and thus help the programmer to choose an efficient data distribution. For our experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), implemented in
Figure 17: Process times for the mesh-spectral Poisson solver application implemented in Fortran with MPI, running on the specified number of nodes on the IBM SP2 (without the crossbar switch) and a network of 166 MHz Pentiums (communicating over 100 Mbps Ethernet). Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 19 for corresponding table.

<table>
<thead>
<tr>
<th></th>
<th>Time (secs) on n nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 1</td>
</tr>
<tr>
<td>Fortran M</td>
<td></td>
</tr>
<tr>
<td>Expected Elapsed Time</td>
<td>512</td>
</tr>
<tr>
<td>Actual Elapsed Time</td>
<td>521</td>
</tr>
<tr>
<td>MPI</td>
<td></td>
</tr>
<tr>
<td>Expected Elapsed Time</td>
<td>513</td>
</tr>
<tr>
<td>Actual Elapsed Time</td>
<td>520</td>
</tr>
</tbody>
</table>

Table 20: Elapsed times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 18 for corresponding graph.
Figure 18: Elapsed times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 20 for corresponding table.

<table>
<thead>
<tr>
<th>Time (secs) on n nodes</th>
<th>n = 1</th>
<th>n = 4</th>
<th>n = 9</th>
<th>n = 16</th>
<th>n = 25</th>
<th>n = 36</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Process Time</td>
<td>508</td>
<td>142</td>
<td>75</td>
<td>54</td>
<td>49</td>
<td>55</td>
</tr>
<tr>
<td>Actual Process Time</td>
<td>516</td>
<td>143</td>
<td>75</td>
<td>54</td>
<td>55</td>
<td>63</td>
</tr>
<tr>
<td>MPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Process Time</td>
<td>509</td>
<td>134</td>
<td>64</td>
<td>39</td>
<td>28</td>
<td>22</td>
</tr>
<tr>
<td>Actual Process Time</td>
<td>516</td>
<td>135</td>
<td>64</td>
<td>39</td>
<td>27</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 21: Process times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figure 19 for corresponding graph.
Figure 19: Process times for the mesh-spectral Poisson solver application, Fortran M and MPI implementations, running on the specified number of nodes on the IBM SP2, without the crossbar switch. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 21 for corresponding table.

Fortran with MPI and running on an IBM SP2.

**Benchmarking.** The computational and communication benchmark programs appear in Appendix B.2. Different choices of `nxprocs` and `nyprocs` (the dimensions of the process grid) imply different data distributions; for example, if `nxprocs = 1`, data is distributed by columns. To model the effect of varying the data distribution in this way, we can reuse the computational benchmark results in Table 10, but we must rerun the communication benchmark for each choice of `(nxprocs, nyprocs)`. We ran the communication benchmark for the following configurations of `(nxprocs, nyprocs)`: (1,16), (2,8), (4,4), (8,2), and (16,1). Results are given in Table 22.

**Performance Model.** Our performance model for this application is given in Section 4.2, based on the program in Appendix B.2, and is applicable to all data distributions (parameterized by `nxprocs` and `nyprocs`).

**Experimental Results.** For this experiment we used the following values of the program parameters:

\[
\begin{align*}
\text{nsteps} &= 1000 \\
\text{ncheck} &= 10
\end{align*}
\]
and NXPROCS and NYPROCS as described earlier. Table 23 compares predicted with observed times. Again, predicted times agree well overall with actual times. Somewhat surprisingly, the model also predicts that for this application the choice of data distribution has little effect on execution time; nevertheless, this is borne out by observed execution times, suggesting that the model’s predictions can indeed help guide the programmer’s choice of data distribution.

5.5 Performance Analysis and Simulation

In this section, we show that our performance model can be used to simulate the actual program executions. For our experiment, we employ the mesh-spectral Poisson solver application (described in Section 4.2), implemented in Fortran M, running on the IBM SP2.

**Benchmarking.** The computational and communication benchmark programs appear in Appendix B.2. For this experiment, we need not only the average results presented Tables 10 and 11 but also, for each measurement, the maximum and minimum of the values used to compute the average. These values are shown in Tables 24 and 25. For each measurement, the simulation generates a uniform distribution using these minimum and maximum values.

<table>
<thead>
<tr>
<th>(NXPROCS, NYPROCS)</th>
<th>(1, 16)</th>
<th>(2, 8)</th>
<th>(4, 4)</th>
<th>(8, 2)</th>
<th>(16, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{set_mesh}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{blk_start}$</td>
<td>868</td>
<td>1454</td>
<td>1276</td>
<td>1315</td>
<td>1428</td>
</tr>
<tr>
<td>$T_{corr_blk}$</td>
<td>1326</td>
<td>1714</td>
<td>1666</td>
<td>1669</td>
<td>1629</td>
</tr>
<tr>
<td>$T_{bdy_exch}$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>$T_{host}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$T_{data_bounds}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{global_max}$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$T_{intersect}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{local_pos}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{local_global}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{pack}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$T_{unpack}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 22: Results of communication benchmark for the mesh-spectral Poisson solver application, running with the specified data distributions on the IBM SP2 using Fortran and MPI, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.
Table 23: Execution times for the mesh-spectral Poisson solver application, running with the specified data distributions on the IBM SP2 using Fortran and MPI, without the crossbar switch. Problem size is 800 by 800 points and 1000 steps. Times are in seconds.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Min, max times (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{overhead}$</td>
<td>100, 100</td>
</tr>
<tr>
<td>$T_{readConst}$</td>
<td>11, 24</td>
</tr>
<tr>
<td>$T_{init}$</td>
<td>222, 223</td>
</tr>
<tr>
<td>$T_{comp}$</td>
<td>426, 431</td>
</tr>
<tr>
<td>$T_{check_converge}$</td>
<td>133, 138</td>
</tr>
<tr>
<td>$T_{copy_values}$</td>
<td>60, 66</td>
</tr>
<tr>
<td>$T_{output}$</td>
<td>15, 16</td>
</tr>
</tbody>
</table>

Table 24: Results of computational benchmark for the mesh-spectral Poisson solver application, running on a single node of the IBM SP2 using Fortran. Grid size is 800 by 800 points. Times are in milliseconds.

**Performance Model.** Our simulation is based on the performance model given in Section 4.2, and on the program in Appendix B.2.

**Experimental Results.** For this experiment we used the following values of the program parameters:

- $\text{NSTEPS} = 1000$
- $\text{NCHECK} = 10$
- $\text{NXPROCS} = \text{NYPROCS} = \sqrt{n}$
Table 25: Results of communication benchmark for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2 using Fortran M, without the crossbar switch. Grid size is 800 by 800 points. Times are in milliseconds.
where \( n \) is the number of processors (1, 4, 9, 16, 25, or 36). Table 26 and Figures 20 and 21 compare predicted, simulated, and actual execution times. These results indicate that our simulation helps us as developers to predict the performance times about as accurately as the performance model. Although before this experiment we surmised that analysis using closed-form equations would be most useful for calculating worst-case performance, and simulation would be most useful for estimating average-case performance, the nature of this application is such that both techniques are useful for estimating actual performance.

<table>
<thead>
<tr>
<th>( n = 1 )</th>
<th>( n = 4 )</th>
<th>( n = 9 )</th>
<th>( n = 16 )</th>
<th>( n = 25 )</th>
<th>( n = 36 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected Elapsed Time</td>
<td>512</td>
<td>152</td>
<td>91</td>
<td>81</td>
<td>98</td>
</tr>
<tr>
<td>Simulated Elapsed Time</td>
<td>533</td>
<td>142</td>
<td>90</td>
<td>75</td>
<td>100</td>
</tr>
<tr>
<td>Actual Elapsed Time</td>
<td>521</td>
<td>151</td>
<td>91</td>
<td>78</td>
<td>95</td>
</tr>
<tr>
<td>Expected Process Time</td>
<td>508</td>
<td>142</td>
<td>75</td>
<td>54</td>
<td>49</td>
</tr>
<tr>
<td>Simulated Process Time</td>
<td>521</td>
<td>138</td>
<td>73</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>Actual Process Time</td>
<td>516</td>
<td>143</td>
<td>75</td>
<td>54</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 26: Elapsed times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2, without the crossbar switch, using Fortran M. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Figures 20 and 21 for corresponding graphs.
Figure 20: Elapsed times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2, without the crossbar switch, using Fortran M. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 26 for corresponding table.

Figure 21: Process times for the mesh-spectral Poisson solver application, running on the specified number of nodes on the IBM SP2, without the crossbar switch, using Fortran M. Problem size is 800 by 800 grid points and 1000 steps. Times are in seconds. See Table 26 for corresponding table.
6 Conclusions

Our experimental results in Section 5 confirm that execution times as predicated by our model are reasonably close to observed execution times\footnote{In most cases, predicted times are, as expected, conservative compared to observed times. Due to time constraints, we did not pursue the question of why in a few cases predicted times were optimistic. One possible explanation is that our assumptions about repeatability were not as valid as we thought. For example, we assumed that execution times on the IBM SP2 would not be affected by other users of the machine, since we had exclusive control of the particular nodes we were using; this assumption ignores possible contention for machine-wide communication resources and so might be invalid.}. More important, experiments designed to test our model’s ability to predict how various “tuning choices” affect execution time gave encouraging results: The tuning choices that gave the best predicted times also gave the best observed times, though in some cases the differences were slight.

The goal of our performance evaluation methodology was to estimate program execution time with sufficient accuracy to guide programmers in making tuning decisions, and to do this in a way that can be incorporated into application development relatively easily. Based on the experiments described in Section 5, we believe that we have met that goal. We conclude that the model, though simple, can be of practical value in application development. Future work could investigate (i) a wider range of archetypes and (ii) the applicability of the methodology to shared-memory architectures.

How well does the model predict performance? In some cases, actual and expected times agree consistently, especially when computation is more than half of the overall running time, and when a finer grain of benchmarking is used.

How useful is the model for making decisions? As demonstrated in Section 5, the model can help programmers choose between different data partitioning and granularity strategies. We conclude that the model, though simple, can be of use to an application developer.

Future Work: Problem Solving Environments. The performance tools we have described in this paper — including the methodology encapsulated by the workflow in Figure 1, the analytic models, and the accompanying simulation techniques — can be nicely managed when bundled with a Problem Solving Environment (PSE). According to Gallopoulos et al. [GHR94], PSEs provide “a framework that is all things to all people: they solve simple or complex problems, support rapid prototyping or detailed analysis, and can be used in introductory education or at the frontiers of science.” As demonstrated by Cheng and Fox [CF06], integrating parallel programming paradigms in a software environment (in their example, programming the CM5 using the visualization software AVS)
enables application developers to become more productive through useful provided tools.

PSEs represent a way to package a computational solution to a problem with the set of tools and methodologies. Employing these tools and methodologies, a scientist can formulate a problem, solve the problem, optimize the solution through refinement, and analyze the results. The PSE provides a user-friendly environment that is natural to the problem domain; for example, the PSE for Air Quality Models [DC97] provides an integrated user interface for defining, solving, and evaluating smog models. This PSE is integrated with the $3D + T + M^k$ archetype; this suggests that integrating it with the mesh and mesh-spectral archetypes would yield similar benefits. Additionally, the integration of the performance model with the problem solving environment further helps an application developer use an archetype.

This suggests another path for future work as well. The performance model and tools presented in this paper could be applied to other archetypes as well, including the one-deep divide and conquer archetype [MC96], which is a pattern that solves divide and conquer programs while only taking the recursion to a single depth. In addition, the performance analysis work done for the aforementioned air quality models [DM97] could be applied to the framework presented in this paper.

**Future Work: Shared-Memory Architectures.** Most of our experiments with archetype-based application development have targeted distributed-memory architectures. A major advantage of an archetype-based approach to developing applications for such archetypes is that a parallel programming archetype can specifically address one of the things that makes such applications difficult to develop, namely the distribution of data. This advantage could be equally useful for a shared-memory architecture for which data locality is crucial to performance — i.e., a shared-memory architecture that is most effectively used by treating it as a distributed-memory architecture. It would be relatively easy to port existing archetype implementations that use message-passing to such a machine, after which some of the experiments described in this and other papers on archetypes [CMMM95, MC96] could be repeated.

Whether an archetype-based approach has similar benefits when the target architecture supports a shared-memory model without performance penalties is a more difficult question. Massingill [Mas98] describes an archetype-based approach to application development one of whose stages can be converted in a straightforward way to a program for a shared-memory architecture, but again the experimental work focuses on the later stages of the process, whereby the original algorithm becomes a program for a distributed-memory architecture. Future work could experiment with using this process to develop practical applications for shared-memory architectures.
Future Work: Task-Parallel Problems. The performance model in this paper was used to evaluate different data partitions and distributions in SPMD programs written using archetypes. Future work could extend to evaluating the model’s utility in analyzing different decompositions and mappings of task-parallel programs as well.

Future Work: Event-Oriented Problems. The overall goal of any distributed resource management system is the efficient matching of resource providers and requestors. Using events as our solutions’ communication substrate, we can develop distributed control announce-listen algorithms that are both scalable and fault-tolerant [Sch96]. The announce-listen paradigm is used at the messaging layer to assist in resource location, reservation, and scheduling [RRDC97]. We have implemented this messaging facility in Java as global events.

Java Beans provide local events as a mechanism by which a component informs other components that something interesting has happened. These events can be thought of as active messages; for example, a button is pressed at a source, and channeled through an event listener, to trigger a method in an event observer automatically. An event propagates from an event source through an event notifier to one or more event observers that respond to the events as they arrive. The notifier routes the event to the observers using a control list, and observers can ask the notifier to be added or removed from this list without notifying the event source.

We have developed a global event structuring mechanism [CRS98] that is identical to the local event model of Java Beans, except that instead of Java Beans’ referencing an object within a single Java Virtual Machine, we use a global name for the object, employing the Web’s URL convention. Furthermore, because the components of the global event system are distributed, multicast can be used for efficient group communication, instead of Java Beans’ local event point-to-point casting.

Using global events, an event is announced by a source object in one virtual machine, and notifiers for that event in other virtual machines anywhere on the Internet listen for the event and forward it to the appropriate (distributed) observers. Unlike the group communication in virtual synchrony [BvR94], it is not necessary for the event sources to know at any point who the event observers will be. Our global event model is useful not only to distribute events and the objects that use them, but also to compose event notifiers, to filter using predicates, and to provide security using access control lists at the event notifier level.

There are several advantages to using global events and soft state. The announce-listen paradigm is fault-resilient [Sch96] that is, if a resource provider goes away, the system adapts dynamically to continue to meet the requests of the consumers. Furthermore, systems constructed using global events and multicast
are compositional and scalable: providers and consumers can add or remove
themselves at any point dynamically. Unfortunately, such systems also have the
potential for oscillation; that is, if state changes faster than the communication
updates, then soft state may give a bad estimate of the current system state.

We are currently investigating the tradeoffs between soft state and hard
invariants, between pushing and pulling resource requests and responses, and
between a hierarchy of middlemen and a flat requestor-provider structure. The
performance tools we have described in this paper — including the methodology
encapsulated by the workflow in Figure 1, the analytic models, and the accom-
panying simulation techniques — can be used in conjunction with distributed
programs communicating using events as the messaging facility, and this is an
exciting area worthy of future exploration.

Acknowledgments

Thanks are due to: Mani Chandy for his advice and leadership; Greg Davis
for his help in developing the mesh-spectral archetype implementation; Anita
Marenno, John Langford, and Lena Petrovic for their help in developing the
applications used for our experiments; and Argonne National Laboratory and
Intel Corporation for giving us access to suitable computing facilities.

References

on the Web, 1996.

[BBC+93] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra,
V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. Templates
for the Solution of Linear Systems: Building Blocks for Iterative

[BH93a] P. Brinch Hansen. Model Programs for Computational Science:
A Programming Methodology for Multicomputers. Concurrency:

for Computational Science. Concurrency: Practice and Experience,

The p4 Parallel Programming System. Parallel Computing, 20:547–


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


A Mesh Programs

This appendix contains source code listings of the sequential applications, parallel applications, computational benchmark programs, and communication benchmark programs based on the mesh archetype and referenced in this paper. For more information about the archetype and its library routines, refer to [Mas96]. In reading these programs, observe that:

- **PARAMETER** values containing underscores (e.g., the definition of `NX` in the heat diffusion program) are intended to be replaced with numeric values by a preprocessor.

- **INCLUDE** files `mesh_uparms.h`, `mesh_parms.h`, and `mesh_common.h` (not shown) contain archetype constants and **PARAMETERS**; see [Mas96] for details.

- Routine `mytime` (not shown) samples wall-clock time; routine `mytimediff` (not shown) computes the difference in milliseconds between two such samples.

### A.1 Heat Diffusion

**Sequential Application Program.**

```c
program heat

implicit none
integer NX
parameter (NX=_NX_/)
integer NSTEPS
parameter (NSTEPS=_NSTEPS_/)

integer IUNIT, OUNIT
parameter (IUNIT=11, OUNIT=12)
character(*) OUTNAME
parameter (OUTNAME='seq_heat.OUT')

real uk(1:NX), ukp(1:NX)
integer istarts, istartms, istops, istopms, itimediff
real dx, dt
integer i, int, k

call mytime(istarts, istartms)
```

dx = 1.0/NX
dt = 0.5*dx*dx

C initialize
print*, 'NX = ', NX
print*, 'NSTEPS = ', NSTEPS
open (unit=UNIT, file=OUTNAME, status='unknown',
  access='sequential', form='formatted')
do i=2,NX-1
  uk(i)=0.0
enddo
uk(1)=1.0
uk(NX)=1.0

C time step loop
do k=1,NSTEPS
  do i=2,NX-1
    ukp1(i)=uk(i)+(dt/dx^2)*(uk(i+1)-2*uk(i)+uk(i-1))
  enddo
  do i=2,NX-1
    uk(i)=ukp1(i)
  enddo

C write (UNIT,*) 'timestep ', k
C write (UNIT,25) (uk(i),i=1,NX)
C25       format (4X,E15.5)
endo

write (UNIT,* 'timestep ', nst)
write (UNIT, (4X,8,E15.5)) i, uk(i)
endo
do i=NX-icnt+1,NX
  write (UNIT, (4X,8,E15.5)) i, uk(i)
endo

close(unit=UNIT)
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms,
  & itimediff)
p = put
C
CParallel Application Program.
C-------------------------------------------------------
C program heat
C explicitly solves the 1-D diffusion equation
C make-time parameters:
C    NX = problem size
C    XPROCS = number of grid processes
C    NSTEPS = number of time steps
C-------------------------------------------------------

subroutine hostmain
  include 'mesh_uparms.h'
  include 'meshParms.h'
  include 'mesh_common.h'

58
C moved to archetype file mesh_uparms.h
C integer NX
C parameter (NX=_NX_)
integer NSTEPS
parameter (NSTEPS=_NSTEPS_)

integer IUNIT, OUNIT
parameter (IUNIT=11, OUNIT=12)
character(*) OUTNAME1, OUTNAME2
parameter (OUTNAME1='par_heat', OUTNAME2='.OUT')

real uk(1:NX)
integer istarts, istartms, istops, istopms, itimediff
real dx, dt
integer i, icnt
character*/8/* outname
character*/4/* cbuff
integer cblen
call mytime(istarts, istartms)
dx = 1.0/NX
dt = 0.5*dx*dx

C initialize
print*, 'NX = ', NX
print*, 'XPROCS = ', XPROCS
print*, 'NSTEPS = ', NSTEPS

C build output filename
if (XPROCS .le. 9) then
  write (cbuff, '(/3i/1./1/)') XPROCS
  cblen=1
else
  write (cbuff, '(/3i/2./2/)') XPROCS
  cblen=2
endif
outname = OUTNAME1 // cbuff(1:cblen) // OUTNAME2
open (unit=OUNIT, file=outname, status='unknown',
     access='sequential', form='formatted')
do i=2,NX-1
  uk(i)=0.0
enddo
uk(1)=1.0
uk(NX)=1.0
call mesh_HtoG_host(uk)

C time step loop
do k=1,NSTEPS
  call mesh_GtoH_host(uk)
  call write (OUNIT,*) 'timestep ', k
  write (OUNIT, 25) (uk(i),i=1,NX)
  format (4X,E15.5)
enddo

C call mesh_GtoH_host(uk)
write (OUNIT,*) 'timestep ', NSTEPS
icnt=min(5,NSTEPS/2)
do i=1,icnt
  write (OUNIT, '(/4X,18,E15.5/)') i, uk(i)
enddo
do i=NX-icnt+1,NX
 subroutine gridmain

 include 'mesh_uparms.h'
 include 'meshParms.h'
 include 'mesh_common.h'

 C moved to archetype file mesh_uparms.h

 integer NX
 parameter (NX=NX_)
 integer NSTEPS
 parameter (NSTEPS=NSTEPS_)

 real uk(IXLO:IXHI),ukp1(IXLO:IXHI)
 integer istarts, istartms, istops, istopms, itimediff
 logical iempty
 integer i, k
 real dx, dt

call mytime(istarts, istartms)
dx = 1.0/NX
dt = 0.5*dx*dx

 C initialize
 call mesh_HtoG_grid(uk)
call xintersect(2,NX-1,istart,iend,iempty)

 C time step loop
 do k=1,NSTEPS
   call mesh_update_bdry(uk)
   do i=istart, iend
     ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
   enddo
   do i=istart, iend
     uk(i)=ukp1(i)
   enddo
 C call mesh_GtoH_grid(uk)
 enddo
 call mesh_GtoH_grid(uk)

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
$ print*, 'process :, iproc, ': ', itimediff
end
Computational Benchmark Program.

program heat

C explicitly solves the 1-D diffusion equation

implicit none
integer NX
parameter (NX=_NX_)
integer NSTEPS
parameter (NSTEPS=_NSTEPS_)

integer IUNIT, OUNIT
parameter (IUNIT=1, OUNIT=12)
character(*) OUTNAME1, OUTNAME2
parameter (OUTNAME1='bench_heat', OUTNAME2='OUT')

real uk(1:NX), ukp(1:NX)
integer istarts, istartms, istops, istopms, timediff
real dx, dt
integer k, klim, i, i cnt
character*80 outname
character*4 cbuff
integer cblen

dx = 1.0/NX
dt = 0.5*dx*dx

print*, 'NX = ', NX
print*, 'NSTEPS = ', NSTEPS

C-------benchmark initialization
call mytime(istarts, istartms)
klim = 10000
do k = 1, klim
    C initialize
    do i = 2, NX-1
        uk(i)=0.0
    enddo
    uk(1)=1.0
    uk(NX)=1.0
    enddo
    call mytime(istarts, istartms)
    call mytimediff(istarts, istartms, istops, istopms, timediff)
    print*, 'initialization: ', klim, timediff,
    print*, dble(timediff)/dble(klim)

C-------benchmark time step
    call mytime(istarts, istartms)
    C time step loop
    klim = 10000
    do k = 1, klim
C do k=1,NSTEPS
  do i=2,NX-1
    ukp1(i)=uk(i)+(dt/(dx*dx))*(uk(i+1)-2*uk(i)+uk(i-1))
  enddo
  do i=2,NX-1
    uk(i)=ukp1(i)
  enddo
C print 'timestep ', k
C print 25,(uk(i),i=1,NX)
C25 format (4X,E15.5)
C enddo
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
$ print*, 'computation: ', klim, itimediff,$
$ dble(itimediff)/dble(klim)
c------benchmark output operation
call mytime(istarts, istartms)
klim = 10
  do k = 1, klim
    build output filename (dummy version)
    write (cbuff, '(3i1.1)') 0
cblen=1
    outname = OUTNAME1 // OUTNAME2
c open and write
    open (unit=OUNIT, file=outname, status='unknown',
    access='sequential', form='formatted')
    write (OUNIT,*),(i,uk(i))
    do i=NX-icnt+1,NX
      write (OUNIT,*),(i,uk(i))
    enddo
    close(unit=OUNIT)
  enddo
  call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
$ print*, 'output: ', klim, itimediff,$
$ dble(itimediff)/dble(klim)
end
C-----------------------------------------------------------

Communication Benchmark Program.
C-----------------------------------------------------------
C program bench_mesh
C written by adam rifkin, adaman@cs.caltech.edu, 1996
C modified by berna massingill
C
C calculates times for the archetype library routines
C
make-time parameters:
C  MX = problem size
C  IPROCS = number of grid processes
C  NTIMES = number of times to execute comm. routines
C
C may also need to adjust loops bounded by klim
subroutine hostmain

include 'mesh_uparms.h'
include 'mesh_parms.h'
include 'mesh_common.h'

C moved to archetype file mesh_uparms.h
C integer NX
C parameter (NX=_NX_)
integer NTIMES
parameter (NTIMES=_NTIMES_)

real uk(1:NX)
integer istarts, istartms, istops, istopms, itimediff

print*, 'NX= ', NX
print*, 'IPROCS= ', IPROCS
print*, 'NTIMES= ', NTIMES

C----- benchmark mesh_GtoH_host
    call mytime(istarts, istartms)
klim = NTIMES
    do k=1,klim
        call mesh_GtoH_host(uk)
    enddo
    call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms,
    $ itimediff)
    print*, 'mesh_GtoH_host: ', klim, itimediff,
    $ dble(itimediff)/dble(klim)

C----- benchmark mesh_HtoG_host
    call mytime(istarts, istartms)
klim = NTIMES
    do k=1,klim
        call mesh_HtoG_host(uk)
    enddo
    call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms,
    $ itimediff)
    print*, 'mesh_HtoG_host: ', klim, itimediff,
    $ dble(itimediff)/dble(klim)
end

subroutine gridmain

include 'mesh_uparms.h'
include 'mesh_parms.h'
include 'mesh_common.h'

C moved to archetype file mesh_uparms.h
C integer NX
C parameter (NX=_NX_)
integer NTIMES
parameter (NTIMES=_NTIMES_)

real uk(iIL0:iIXI), ukpi(iIL0:iIXI)
integer istarts, istartms, istops, istopms, itimediff
C------ benchmark mesh_GtoH_grid
    call mytime(istarts, istartms)
    klim = NTIMES
    do k=1,klim
       call mesh_GtoH_grid(uk)
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms,
        itimediff)
    print*, 'mesh_GtoH_grid process ', iproc, ': ', klim,
        itimediff, dble(itimediff)/dble(klim)
C------ benchmark mesh_HtoG_grid
    call mytime(istarts, istartms)
    klim = NTIMES
    do k=1,klim
       call mesh_HtoG_grid(uk)
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms,
        itimediff)
    print*, 'mesh_HtoG_grid process ', iproc, ': ', klim,
        itimediff, dble(itimediff)/dble(klim)
C------ benchmark mesh_update_bdry
    call mytime(istarts, istartms)
    klim = NTIMES
    do k=1,klim
       call mesh_update_bdry(uk)
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms,
        itimediff)
    print*, 'mesh_update_bdry process ', iproc, ': ', klim,
        itimediff, dble(itimediff)/dble(klim)
C------ benchmark xintersect
    call mytime(istarts, istartms)
    klim = 1000000
    do k=1,klim
        call xintersect(2,NX-1,ibegin,iend,iempty)
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms,
        itimediff)
    print*, 'xintersect process ', iproc, ': ', klim,
        itimediff, dble(itimediff)/dble(klim)

end

C------------------------------------------------------------------
A.2 Poisson Solver

Sequential Application Program.

C-------------------------------------------------------------------------------
C  program poisson
C  uses Jacobi relaxation to solve the Poisson equation
C  make-time parameters:
C     NI, NY = problem size
C     MAXSTEPS = maximum number of iterations
C-------------------------------------------------------------------------------

program poisson

integer NI
integer NY
parameter (NI=NI_)
parameter (NY=NY_)

real H
parameter (H=0.05)

C how often to check for convergence
integer NCHECK
parameter (NCHECK=10)

C convergence criterion
real TOL
parameter (TOL=0.0001)

C maximum number of steps
integer MAXSTEPS
parameter (MAXSTEPS=MAXSTEPS_)

C file units, names
integer IUNIT, OUNIT
parameter (IUNIT=1, OUNIT=2)
character(*) OUTNAME
parameter (OUTNAME='seq_poisson.OUT')

external F
external G
real uk(1:NX,1:NY), ukp(1:NX,1:NY)
real diff, diffmax
integer istarts, istartms, istops, istopms, itimediff

C initialize
  call mytime(istarts, istartms)
  print*, 'NX = ', NX
  print*, 'NY = ', NY
  print*, 'NCHECK = ', NCHECK
  open (unit=OUNIT, file=OUTNAME, status='unknown',
       access='sequential', form='formatted')

C interior points
  do j = 2, NY-1
    do i = 2, NX-1
      uk(i,j) = F(i,j,NX,NY,H)
    enddo
  enddo

C boundary points
  do j = 1, NY
    uk(1,j) = G(1,j,NX,NY,H)
    uk(NX,j) = G(NX,j,NX,NY,H)
  enddo

65
do i = 2, NX-1
    uk(i,1) = G(i,1,NX,NY,H)
    uk(i,NY) = G(i,NY,NX,NY,H)
enddo
C loop until convergence
diffmax = TOL + 1.0
do k=1,MAXSTEPS
C compute new values
do j = 2, NY-1
do i = 2, NX-1
    ukp1(i,j) = 0.25*(H+H+F(i,j,NX,NY,H))
    - * uk(i+1,j) + uk(i,j+1)
    + * uk(i-1,j) + uk(i,j-1)
enddo
C every NOCHECK-th step, recompute convergence test
if (mod(k,NOCHECK) .eq. 0) then
    diffmax = 0.0
    do j = 2, NY-1
        do i = 2, NX-1
            diff = abs(ukp1(i,j) - uk(i,j))
            if (diff .gt. diffmax) diffmax = diff
        enddo
    enddo
endif
C copy new values to old values
do j = 2, NY-1
    do i = 2, NX-1
        uk(i,j) = ukp1(i,j)
    enddo
C check for convergence
if (diffmax .le. TOL) go to 1000
C print results
print 1000
continue
write (UNIT,*) 'NX, NY = ', NX, NY
write (UNIT,*) 'H = ', H
write (UNIT,*) 'tolerance = ', TOL
write (UNIT,*) 'NOCHECK = ', NOCHECK
call Fprint(UNIT)
call Gprint(UNIT)
if (diffmax .le. TOL) then
    write (UNIT,*) 'convergence occurred in ', k, ' steps'
    print*, 'steps = ', k
else
    write (UNIT,*) 'no convergence in ', MAXSTEPS,
    ' steps; ', 'max. difference ', diffmax
    print*, 'steps = ', MAXSTEPS
endif
C do i = 1, NX
c if (NX .gt. 10) write (UNIT,*) ' '
c write (UNIT,9998) (uk(i,j), j = 1, NY)
do i = 1, NX, NX/5
    do j = 1, NY, NY/5
        write (UNIT,9998) i, j, uk(i,j)
    enddo
enddo
write (UNIT,9998) i, NY, uk(i,NY)
write (UNIT,*), '
enddo
do j = 1, NY, NY/5
write (UNIT,9998) j, uk(j)
enddo
write (UNIT,9998) NY, uk(NY)
9998 format (2i10,f8.4)
9999 format (f9.4)
close(unit=UNIT)
call mytime(istops, istopms)
call my timediff(istarts, istartms, istops, istopms, itimediff)
print*, 'sequential time:', itimediff
end
real function F(i,j,nx,ny,h)
integer i, j, nx, ny
real h
F = 0.0
end
subroutine Fprint(ounit)
integer ounit
write (ounit,*) 'F(i,j) = 0.0'
end
real function G(i,j,nx,ny,h)
integer i, j, nx, ny
real h
G = (i+j)*h
end
subroutine Gprint(ounit)
integer ounit
write (ounit,*) 'G(i,j) = (i+j)*h'
end
Parallel Application Program.

C program poisson
C uses Jacobi relaxation to solve the Poisson equation
C
C make-time parameters:
C NX, NY = problem size
C XPROCS, YPROCS = number of grid processes
C MAXSTEPS = maximum number of iterations
C
C integer NX
C integer NY

67
C parameter (NX=NX)
C parameter (NY=NY)

C subroutine hostmain

include 'mesh_uparms.h'
include 'mesh_parms.h'
include 'mesh_common.h'

real H
parameter (H=0.05)

C how often to check for convergence
integer NCHECK
parameter (NCHECK=10)

C convergence criterion
real TOL
parameter (TOL=0.00001)

C maximum number of steps
integer MAXSTEPS
parameter (MAXSTEPS=MAXSTEPS)

C file units, names
integer IUNIT, OUNIT
parameter (IUNIT=11, OUNIT=12)
character(*) OUTNAME1, OUTNAME2
parameter (OUTNAME1='par_poisson', OUTNAME2='.OUT')

external F
external G

real uk(1:NX,1:NY)
real difflocal, diffmax
integer istarts, istartms, istops, stopms, timediff
character(*8) outname
character(*4) cbuff1, cbuff2
integer cblen1, cblen2

C initialize

call mytime(istarts, istartms)
print*, 'NX = ', NX
print*, 'NY = ', NY
print*, 'XPROCS = ', XPROCS
print*, 'YPROCS = ', YPROCS
print*, 'MYCHECK = ', MYCHECK

C build output filename

if (XPROCS .le. 9) then
    write (cbuff1, '(3i1.1)') XPROCS
    cblen1=1
else
    write (cbuff1, '(3i2.2)') XPROCS
    cblen1=2
endif
if (YPROCS .le. 9) then
    write (cbuff2, '(3i1.1)') YPROCS
    cblen2=1
else
    write (cbuff2, '(3i2.2)') YPROCS
    cblen2=2
endif
outname = OUTNAME1 // cbuff1(1:cblen1) // '_'
    // cbuff2(1:cblen2) // OUTNAME2
open (unit=OUNIT, file=outname, status='unknown',
interior points
do j = 2, NY-1
do i = 2, NX-1
   uk(i,j) = F(i,j,NX,NY,H)
enddo
enddo

boundary points

do j = 1, NY
   uk(1,j) = G(1,j,NX,NY,H)
   uk(NX,j) = G(NX,j,NX,NY,H)
enddo

do i = 2, NX-1
   uk(i,1) = G(i,1,NX,NY,H)
   uk(i,NY) = G(i,NY,NX,NY,H)
enddo

move to grid
call mesh_HtoG_host(uk)

loop until convergence
diffmax = TOL + 1.0

do k = 1, MAXSTEPS

compute new values
(computation in grid only)
every NCHECK-th step, recompute convergence test
if (mod(k,NCHECK) .eq. 0) then
   (computation in grid only)
call mesh_merge_real_maxabs(1, difflocal, diffmax)
endif

copy new values to old values
(computation in grid only)

check for convergence
if (diffmax .le. TOL) go to 1000

enddo

print results

continue

1000

call mesh_GtoH_host(uk)
write (UNIT,*) 'NX, NY = ', NX, NY
write (UNIT,*) 'H = ', H
write (UNIT,*) 'tolerance = ', TOL
write (UNIT,*) 'NCHECK = ', NCHECK

call Fprint(UNIT)
call Gprint(UNIT)

if (diffmax .le. TOL) then
   write (UNIT,*) 'convergence occurred in ', k, ' steps'
   print*, 'steps = ', k
else
   write (UNIT,*) 'no convergence in ', MAXSTEPS,
   ' steps; ', 'max. difference ', diffmax
   print*, 'steps = ', MAXSTEPS
endif

do i = 1, NX
   if (NX .gt. 10) write (UNIT,*) ' '
   write (UNIT,9999) (uk(i,j), j = 1, NY)
enddo

do i = 1, NX, NX/5
   do j = 1, NY, NY/5
      write (UNIT,9998) i, j, uk(i,j)
   enddo
SUBROUTINE GRIDMAIN
  INCLUDE 'MESH_UPARMS.H'
  INCLUDE 'MESH_PARMS.H'
  INCLUDE 'MESH_COMMON.H'
  REAL H
  PARAMETER (H = 0.05)

  INTEGER NCHECK
  PARAMETER (NCHECK = 10)

  REAL TOL
  PARAMETER (TOL = 0.00001)

  INTEGER MAXSTEPS
  PARAMETER (MAXSTEPS = _MAXSTEPS_)

  EXTERNAL F
  EXTERNAL G
  REAL UK(IXLO:IXHI, IYLO:IYHI), UKP(1:NKL1SIZE, 1:NYL1SIZE)
  REAL DIFF, DIFFLOCAL, DIFFMAX
  LOGICAL IEMPTY, JEMPTY
  INTEGER ISTARTS, ISTARTMS, ISTOP, ISTOPMS, ITIMEDIFF

  C initialization
  CALL MYTIME(ISSTARTS, ISTARTMS)
  C (initialization in host only)
  CALL MESH_HTOG_GRID(UK)

  C compute loop bounds
  CALL XINTERSECT(2, NX-1, ISTART, IEND, IEMPTY)
  CALL YINTERSECT(2, NY-1, JSTART, JEND, JEMPTY)

  C compute starting global indices (of local section)
  CALL IGLOBAL(IPROCX, ISTART, ISTARTG)
  CALL JGLOBAL(IPROCY, JSTART, JSTARTG)

  C compute offsets (global index = offset * local index)
  IOFFSET = ISTARTG - ISTART
  JOFFSET = JSTARTG - JSTART

  C loop until convergence
  DIFFMAX = TOL * 1.0
  DO K = 1, MAXSTEPS
    CALL F(JOFFSET, IOFFSET)
    CALL G(JOFFSET, IOFFSET)
  ENDDO
  DIFF = DIFF + DIFFLOCAL
  DIFFMAX = MAX(DIFFMAX, DIFF)
  IF (DIFFMAX > TOL) THEN
    ISTOP = ISTOP + 1
    CONTINUE
  ELSE
    ISTOP = ISTOP + 1
    CONTINUE
  ENDIF
END
call mesh_update_bdry(uk)

C compute new values
do j = jstart, jend
   do i = istart, iend
      ukp1(i,j) = 0.25*(H/H/F(ioffset+i,joffset+j,NX,NY,H) - uk(i-1,j) + uk(i,j-1) - uk(i+1,j) + uk(i,j+1))
   enddo
endo
done

C every NCHECK-th step, recompute convergence test
if (mod(k,NCHECK) .eq. 0) then
   difflocal = 0.0
   do j = jstart, jend
      do i = istart, iend
         diff = abs(ukp1(i,j) - uk(i,j))
         if (diff .gt. difflocal) difflocal = diff
      enddo
endo
endif

call mesh_merge_real_maxabs(1, difflocal, difffmax)

C copy new values to old values
do j = jstart, jend
   do i = istart, iend
      uk(i,j) = ukp1(i,j)
   enddo
endo

call mesh_update_bdry(uk)

C check for convergence
if (difffmax .le. TOL) go to 1000
endo

C print results
1000 continue
C (printing in host only)
call mesh_GtoH_grid(uk)
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, $ itimediff)
print*, 'process ', iproc, ': ', itimediff
end

C---------------------------------------------------

real function F(i,j,nx,ny,h)
integer i, j, nx, ny
real h
F = 0.0
end

subroutine Fprint(ounit)
integer ounit
write (ounit,*) 'F(i,j) = 0.0'
end

C---------------------------------------------------

real function G(i,j,nx,ny,h)
integer i, j, nx, ny
real h
G = (i+j)*h
end
subroutine Gprint(ounit)
  integer ounit
  write (ounit,*,'(i,j)=(i+j)*H')
end

Computational Benchmark Program.

program poisson
  C uses Jacobi relaxation to solve the Poisson equation
  integer NX
  integer NY
  parameter (NX=_NX_/)
  parameter (NY=_NY_/)

  real H
  parameter (H=0.05)

  integer NCHECK
  parameter (NCHECK=10)

  real TOL
  parameter (TOL=0.00001)

  integer MAXSTEPS
  parameter (MAXSTEPS=_MAXSTEPS_/)

  integer IUNIT, OUNIT
  parameter (IUNIT=11, OUNIT=12)

  character*(80) OUTNAME1, OUTNAME2
  parameter (OUTNAME1='bench_poisson', OUTNAME2='.OUT')

  external F
  external G

  real uk(1:NX,1:NY), ukp1(1:NX,1:NY)
  real diff, diffmax

  integer istarts, istartms, istops, istopms, itimediff

  call mytime(istarts, istartms)

  print*, 'NX = ', NX
  print*, 'NY = ', NY
C--------benchmark initialization
klim = 1000

C initialize
C interior points
    do j = 2, NY-1
      do i = 2, NX-1
        uk(i,j) = F(i,j,NX,NY,H)
      enddo
    enddo
C boundary points
    do j = 1, NY
      uk(1,j) = G(1,j,NX,NY,H)
      uk(NX,j) = G(NX,j,NX,NY,H)
    enddo
    do i = 2, NX-1
      uk(i,1) = G(i,1,NX,NY,H)
      uk(i,NY) = G(i,NY,NX,NY,H)
    enddo
enddo

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
$ print*, 'initialization: ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C loop until convergence
C
C-------------benchmark computation
igoffset=0
jgoffset=0
call mytime(istarts, istartms)
klim = 1000

do k = 1, klim
C compute new values
    do j = 2, NY-1
      do i = 2, NX-1
        ukp1(i,j) = 0.25*(H*H*F(igoffset+i,jgoffset+j,NX,NY,H) -
          * uk(i-1,j) + uk(i,j-1) -
          * uk(i+1,j) + uk(i,j+1) )
      enddo
    enddo
C-------------benchmark computation
call mytime(istarts, istartms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
$ print*, 'computation: ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C every NCHECK-th step, recompute convergence test
C
C-------------benchmark convergence test
if (mod(k,NCHECK) .eq. 0) then
    call mytime(istarts, istartms)
klim = 1000

do k = 1, klim
C
C-------------benchmark convergence test
    diffmax = 0.0
    do j = 2, NY-1
      do i = 2, NX-1
        diff = abs(ukp1(i,j) - uk(i,j))
      enddo
    enddo
    if (diff .gt. diffmax) diffmax = diff

enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'convergence test:', klim, itimediff,
& dble(itimediff)/dble(klim)
c endif

C-----benchmark copy
call mytime(istarts, istartms)
klim = 1000
do k = 1, klim
  copy new values to old values
  do j = 2, NY-1
    do i = 2, NX-1
      uk(i, j) = ukp1(i, j)
    enddo
  enddo
enddo

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'copy:', klim, itimediff,
& dble(itimediff)/dble(klim)
c
check for convergence
c if (diffmax .le. TOL) go to 1000
c endif

C-----benchmark output
call mytime(istarts, istartms)
klim = 100
do k = 1, klim
  build output filename (dummy version)
  write (cbuff1, '(3i1)') 0
  cblen1 = 1
  write (cbuff2, '(3i1)') 0
  cblen2 = 1
  outname = OUTNAME1 // OUTNAME2
  open and write
  open (unit=OUNIT, file=outname, status='unknown',
        access='sequential', form='formatted')
  write (OUNIT,*) 'NX, NY = ', NX, NY
  write (OUNIT,*) 'H = ', H
  write (OUNIT,*) 'tolerance = ', TOL
  write (OUNIT,*) 'NCHECK = ', NCHECK
  call Fprint(OUNIT)
call Gprint(OUNIT)
  if (diffmax .le. TUL) then
    write (OUNIT,*)
    print*, 'convergence occurred in ', k, ' steps'
c else
    write (OUNIT,*) 'no convergence in ', MAXSTEPS,
    print*, 'steps:', 'max difference', diffmax
c endif
  do i = 1, NX
c    if (NX .gt. 10) write (OUNIT,*)
c    write (OUNIT,9999) (uk(i, j), j = 1, NY)
  do i = 1, NX
    do j = 1, NY

74
write (UNIT,9998) i, j, uk(i,j)
enddo
write (UNIT,9998) i, NY, uk(i,NY)
write (UNIT,9999)

enddo
do j = 1, NY, NY/5
write (UNIT,9998) NX, j, uk(NX,j)
enddo
write (UNIT,9998) NX, NY, uk(NX,NY)
format(2110,f8.4)
9996
format(679.4)
close(unit=UNIT)
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms)
$ itimediff
$ print* ,'output: ', klim, itimediff,
$ dble(itimediff)/dble(klim)
end

C------------------------------------------------------------------------------------------
real function F(i,j,nx,ny,h)
integer i, j, nx, ny
real h
F = 0.0
end

subroutine Fprint(ounit)
integer ounit
write (ounit,*),'F(i,j) = 0.0'
end

C------------------------------------------------------------------------------------------
real function G(i,j,nx,ny,h)
integer i, j, nx, ny
real h
G = (i+j)*h
end

subroutine Gprint(ounit)
integer ounit
write (ounit,*),'G(i,j) = (i+j)*H'
end

C------------------------------------------------------------------------------------------

Communication Benchmark Program.
C------------------------------------------------------------------------------------------
C program bench_mesh
C written by adam rifkin, adam@cs.caltech.edu, 1996
C modified by berna massingill
C calculates times for the archetype library routines
C make-time parameters:
C     NX, NY = problem size
C XPROCS, YPROCS = number of grid processes
C NTIMES = number of times to execute comm. routines
C
C may also need to adjust loops bounded by klim
C
C------------------------------------------------------------------

subroutine hostmain

include /'mesh_uparms.h'/
include /'mesh_Parms.h'/
include /'mesh_common.h'/

integer NTIMES
parameter (NTIMES= _NTIMES_ )

real uk(1:NX,1:NY)
real difflocal, diffmax
integer istarts, iarrival, istops, iarrivalms, itimediff

print*, 'NX= ', NX
print*, 'NY= ', NY
print*, 'XPROCS= ', XPROCS
print*, 'YPROCS= ', YPROCS
print*, 'NTIMES= ', NTIMES

C/-/-/-/-/- benchmark mesh_GtoH_host

call mytime (istarts, iarrival)
klim = NTIMES
do k=1,klim
  call mesh_GtoH_host(uk)
enddo

call mytime (istarts, iarrivalms)
call mytimediff (istarts, iarrivalms, istops, iarrivalms, itimediff)
print*, 'mesh_GtoH_host: ', klim, itimediff,
  dble(itimediff)/dble(klim)

C/-/-/-/-/- benchmark mesh_HtoG_host

call mytime (istarts, iarrival)
klim = NTIMES
do k=1,klim
  call mesh_HtoG_host(uk)
enddo

call mytime (istarts, iarrivalms)
call mytimediff (istarts, iarrivalms, istops, iarrivalms, itimediff)
print*, 'mesh_HtoG_host: ', klim, itimediff,
  dble(itimediff)/dble(klim)

C/-/-/-/-/- benchmark mesh_merge_real_maxabs

call mytime (istarts, iarrival)
klim = NTIMES
do k=1,klim
  call mesh_merge_real_maxabs(1, difflocal, diffmax)
enddo

call mytime (istarts, iarrivalms)
call mytimediff (istarts, iarrivalms, istops, iarrivalms, itimediff)
print*, 'mesh_merge_real_maxabs: ', klim, itimediff,
  dble(itimediff)/dble(klim)
subroutine gridmain

include 'mesh_uparms.h'
include 'mesh_parms.h'
include 'mesh_common.h'

integer NTIMES
parameter (NTIMES=_NTIMES_)

real UK(ILO:IHI, IYLO:IYHI)
real difflocal, diffmax
logical iempty, jempty
integer istarts, istartms, istops, istopms, itimediff

C---- benchmark mesh_GtoH_grid

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
   call mesh_GtoH_grid(uk)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms,
& itimediff)
print*, 'mesh_GtoH_grid process ', iprocx, iprocy, ': ', klim,
& itimediff
& dble(itimediff)/dble(klim)

C---- benchmark mesh_HtoG_grid

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
   call mesh_HtoG_grid(uk)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms,
& itimediff)
print*, 'mesh_HtoG_grid process ', iprocx, iprocy, ': ', klim,
& itimediff
& dble(itimediff)/dble(klim)

C---- benchmark mesh_merge_real_maxabs

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
   call mesh_merge_real_maxabs(1, difflocal, diffmax)
enddo
call mytime(istarts, istartms)
call mytimediff(istarts, istartms, istops, istopms,
& itimediff)
print*, 'mesh_merge_real_maxabs process ', iprocx, iprocy,
& ': ', klim,
& itimediff
& dble(itimediff)/dble(klim)

C---- benchmark mesh_update_bdry

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
   call mesh_update_bdry(uk)
   call mytime(istarts, istopms)
   call mytimediff(istarts, istartms, istops, istopms, itimediff)
   print*, 'mesh_update_bdry process ', iproc, iprocy, ': ', klim,
   itimediff,
   dble(itimediff)/dble(klim)
enddo

C------ benchmark xintersect

call mytime(istarts, istartms)
   klim = 100000
   do k=1,klim
      call xintersect(2,NX-1,i1,i2,iempty)
   enddo
   call mytime(istarts, istopms)
   call mytimediff(istarts, istartms, istops, istopms, itimediff)
   print*, 'xintersect process ', iproc, iprocy, ': ', klim,
   itimediff,
   dble(itimediff)/dble(klim)

C------ benchmark yintersect

call mytime(istarts, istartms)
   klim = 100000
   do k=1,klim
      call yintersect(2,NY-1,j1,j2,jempty)
   enddo
   call mytime(istarts, istopms)
   call mytimediff(istarts, istartms, istops, istopms, itimediff)
   print*, 'yintersect process ', iproc, iprocy, ': ', klim,
   itimediff,
   dble(itimediff)/dble(klim)

C------ benchmark iglobal

call mytime(istarts, istartms)
   klim = 100000
   do k=1,klim
      call iglobal(iproc,1,i1)
   enddo
   call mytime(istarts, istopms)
   call mytimediff(istarts, istartms, istops, istopms, itimediff)
   print*, 'iglobal process ', iproc, iprocy, ': ', klim,
   itimediff,
   dble(itimediff)/dble(klim)

C------ benchmark jglobal

call mytime(istarts, istartms)
   klim = 100000
   do k=1,klim
      call jglobal(iproc,1,j1)
   enddo
   call mytime(istarts, istopms)
   call mytimediff(istarts, istartms, istops, istopms, itimediff)
   print*, 'jglobal process ', iproc, iprocy, ': ', klim,
   itimediff,
B Mesh-Spectral Programs

This appendix contains source code listings of the sequential applications, parallel applications, computational benchmark programs, and communication benchmark programs based on the mesh-spectral archetype and referenced in this paper. For more information about the archetype and its library routines, refer to [DM96]. In reading these programs, observe that:

- INCLUDE files arch_uparms.h and arch_parms.h (not shown) contain archetype constants and parameters; see [DM96] for details. Other INCLUDE files not shown (for example, par_fft_header.h and par_fftWrapper.h in the parallel 2D FFT program) are generated by the archetype’s utility program for generating header files; see [DM96] for details and examples.

- Routine mytime (not shown) samples wall-clock time; routine mytimediff (not shown) computes the difference in milliseconds between two such samples.

B.1 Two-dimensional Fast Fourier Transform

Sequential Application Program.

```
program main
implicit none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'seq_fft_header.h'
processors (MAXPROCS)
call spawn (procs)
cend
```

```
cexample of mesh/spectral archetype
c2D FFT
c make-time parameters:
cMX, MY = dimensions of grid
c
```

```c
main program
c```

```c
program main
implicit none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'seq_fft_header.h'
processors (MAXPROCS)
call spawn (procs)
cend
```
process-main program

subroutine procmain()
implicit none
include 'arch/uparms.h'
include 'arch/pars.h'
include 'seq_fft_header.h'
integer nrepeats
parameter (nrepeats=10)
integer IUNIT, OUNIT
parameter (IUNIT=1, OUNIT=2)
character(*) FFT_IN
parameter (FFT_IN='fft.IN')
character(*) FFT_OUT
parameter (FFT_OUT='seq_fft.OUT')
external readproc, writeproc
integer procnum, proctype, datasize
integer istarts, istartms, istops, istopms, itimediff

arguments for cfft99:
cfft99(a, work, trigs,ifax, inc, jump, n, lot, isign)
   a contains input/output of FFT (a set of vectors)
   work is a work array
   trigs,ifax contain constants for cfft99 (generated by
   subroutine cftfax)
   inc is stride between vector elements
   jump is between successive vectors
   n is size of each vector
   lot is how many vectors
   isign indicates forward/inverse transform

integer ifaxnx(3), ifaxny(3)
real trigsnx(2*nx_dg,1), trigsny(2*ny_dg,1)
integer inc, jump, n, lot, m
local section of distributed grid -- type is really complex,
declared double precision for compatibility with archetype
double precision arr(lclsize_dg)
double precision arr(nx_dg*ny_dg)
complex work(nx_dg*ny_dg)
call mytime(istarts, istartms)
call local_pos(proctype, procnum, -1)
if (procnum.eq.IDPROC) then
   print*, 'procs = ', procs
   print*, 'nx_dg = ', nx_dg1
   print*, 'ny_dg = ', ny_dg1
   print*, 'nrepeats = ', nrepeats
endif
do m = 1, nrepeats
cset up grid and read input data; open output file
call wrapper(id_dgl, id_row, DOUBLEPRECISION_T, arr, SETMES)
c if (procnum.eq.IOPROC) then
  open(unit=IUNIT, file=FFT_IN, form='unformatted')
c endif

c call read_mesh(readproc, id_dgl, IUNIT, arr)
call readproc(0, IUNIT, 1,1,1,1, ny_dgl,1,1,1,1,
- mx_dgl, ny_dgl,1,1, arr)
c if (procnum.eq.IOPROC) then
close(IUNIT)
c endif

c initialize for fft

c call cftfax(nx_dgl, ifaxnx, trigsnx)
call cftfax(ny_dgl,ifaxny, trigsny)

c do fft by rows

c call local_pos(procxyz,procnum,id_dgl)
call data_widths(procxyz,false,datasize,id_dgl)
c inc = datasize(1)
  inc = nx_dgl
  jump = 1
  n = ny_dgl
  lot = datasize(1)
  lot = mx_dgl
  call cfft99(arr, work, trigsny, ifaxny, inc, jump, n, lot, -1)

c redistribute data

c call wrapper(id_dgl,id_col,DOUBLEPRECISION,T,arr,REDISTRIBUTEDATA)

-----benchmark fft by columns

c do fft by columns

c call local_pos(procxyz,procnum,id_dgl)
call data_widths(procxyz,false,datasize,id_dgl)
  inc = 1
  jump = mx_dgl
  n = nx_dgl
  lot = datasize(2)
  lot = ny_dgl
  call cfft99(arr, work, trignx, ifaxnx, inc, jump, n, lot, -1)

c redistribute data

c call wrapper(id_dgl,id_row,DOUBLEPRECISION,T,arr,REDISTRIBUTEDATA)

c write output data

c if (procnum.eq.IOPROC) then
  open(unit=OUNIT, file=FFT_OUT, form='unformatted')
c endif

c call write_mesh(writ proc, id_dgl, OUNIT, arr)
call writeproc(0, OUNIT, 1,1,1, mx_dgl,1,ny_dgl,1,1,1, 
- nx_dgl, ny_dgl,1,1, arr)
c if (procnum.eq.IOPROC) then
close(OUNIT)
c endif
```
c end of 'do m = 1, nrepeats' loop
enddo

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print", 'sequential time: ', itimediff

c return
end

****
c subroutine for reading input data (one local section at a time)
****

subroutine readproc(ignum, iunit, ipx, ipy, ipz, ixlo, izhi, iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, rarr)
implicit none
integer ignum, iunit, x, y, z
integer ipx, ipy, ipz
integer ixlo, izhi, iylo, iyhi, izlo, izhi
integer nxlcl, nylcl, nzlcl
complex rarr(ixlo:izhi, iylo:iyhi, izlo:izhi)
do x = 1, nxlcl
  do y = 1, nylcl
    do z = 1, nzlcl
      read(iunit, end=10) rarr(x, y, z)
    enddo
  enddo
endo
10 return
end

****
c subroutine for writing input data (one local section at a time)
****

subroutine writeproc(ignum, ounit, ipx, ipy, ipz, ixlo, izhi, iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, warr)
implicit none
integer ignum, ounit
integer ipx, ipy, ipz, x, y, z
integer ixlo, izhi, iylo, iyhi, izlo, izhi
integer nxlcl, nylcl, nzlcl
complex warr(ixlo:izhi, iylo:iyhi, izlo:izhi)
do x = 1, nxlcl
  do y = 1, nylcl
    do z = 1, nzlcl
      write(ounit) warr(x, y, z)
    enddo
  enddo
endo
10 return
end
```
automatically-generated wrapper program to pack arrays
and call set_mesh or redistribute_data

include 'seq_fft_wrap.h'

Parallel Application Program.

example of mesh/spectral archetype
2D FFT
make-time parameters:
NX, NY = dimensions of grid
NPROCS = number of processes

main program

program main
implicit none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'par_fft_header.h'
processors (MAXPROCS)
call spawn (procs)
end

process-main program

subroutine procmain()
implicit none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'par_fft_header.h'
integer nrepeats
parameter (nrepeats=10)
integer UNIT, OUNIT
parameter (UNIT=11, OUNIT=12)
character(*) FFT_IN
parameter (FFT_IN='fft_IN')
character(*) FFT_OUT1, FFT_OUT2
parameter (FFT_OUT1='par_fft_.FFT_OUT1'.OUT')
external readproc, writeproc
integer procm, procyz(3), datasize(3)
integer istarts, istartms, istops, istopms, itimediff

arguments for fft99:
  fft99(a, work, trigs, ifax, jump, n, lot, isign)
  a contains input/output of FFT (a set of vectors)
  work is a work array
  trigs, ifax contain constants for fft99 (generated by
  subroutine cftfax)
  inc is stride between vector elements
  jump is between successive vectors
  n is size of each vector
  lot is how many vectors
  isign indicates forward/inverse transform

integer ifaxnx(13), ifaxny(13)
real trigsnx(2*nx_dgl), trigsny(2*ny_dgl)
integer inc, jump, n, lot, m

local section of distributed grid -- type is really complex,
declared double precision for compatibility with archetype
do not mix REAL and DOUBLE PRECISION

double precision arr(lclsized_dgl)
complex work(lclsized_dgl)

character*80 fft_out
character*4 cbuff1
integer cblen1

call mytime(istarts, istartms)
call local_pos(procyz, procm, -1)
if (procm.eq.IDPROC) then
  print*, 'procs = ', procs
  print*, 'nx_dgl = ', nx_dgl
  print*, 'ny_dgl = ', ny_dgl
  print*, 'nrepeats = ', nrepeats
endif
do m = 1, nrepeats

c set up grid and read input data
call wrapper(id_dgl, id_row, DOUBLEPRECISION_T, arr, SETMESH)
if (procm.eq.IDPROC) then
  open(unit=IUNIT, file=FFT_IN, form='unformatted')
endif
call read_mesh(readproc, id_dgl, IUNIT, arr)
if (procm.eq.IDPROC) then
  close(IUNIT)
endif
c initialize for fft
call cftfax(nx_dgl, ifaxnx, trigsnx)
call cftfax(ny_dgl, ifaxny, trigsny)
c do fft by rows
call local_pos(procyz, procm, id_dgl)
call data_widths(procyz, false, datasize, id_dgl)
inc = datasize(1)
jump = 1
n = nx_dgl
lot = datasize(1)
call cfft99(arr, work, trigsny, ifaxny, inc, jump, n, lot, -1)

redistribute data
call wrapper(id_dgl, id_col, DOUBLEPRECISION_T, arr, REDISTRIBUTEATA)
do fft by columns
call local_pos(procxyz, procnum, id_dgl)
call data_widths(procxyz, false, datasize, id_dgl)
inc = 1
jump = nx_dgl
n = nx_dgl
lot = datasize(2)
call cfft99(arr, work, trigsnx, ifaxnx, inc, jump, n, lot, -1)

redistribute data
call wrapper(id_dgl, id_row, DOUBLEPRECISION_T, arr, REDISTRIBUTEATA)
write output data
if (procnum.eq.IDPROC) then
  build output filename
  if (procs.le.9) then
    write (chuff1, '(3i1)') procs
    chlen=1
  else
    write (chuff1, '(3i2)') procs
    chlen=2
  endif
  fft_out = FFT_OUT1 // chuff1(1:chlen) // FFT_OUT2
  open(unit=OUNIT, file=fft_out, form='unformatted')
  endif
  call write_mesh(writeproc, id_dgl, OUNIT, arr)
  if (procnum.eq.IDPROC) then
    close(OUNIT)
  endif
end if 'do m = 1, nrepeats' loop
enddo

call mytime(istarts, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'process ', procnum, ': ', itimediff
return
end

-----------------------------------------------------------------------------------

subroutine for reading input data (one local section at a time)
-----------------------------------------------------------------------------------

subroutine readproc(ignum, imit, ipx, ipy, ipz, islo, ixhi,
implicit none
integer ignum, iunit, x, y, z
integer ipx, ipy, ipz
integer ixlo, ixyi, iylo, iyhi, izlo, izhi
integer nxlcl, nylcl, nzlcl
complex rarr(ixlo:ixhi, iylo:iyhi, izlo:izhi)
do x=1,nxlcl
   do y=1,nylcl
      do z=1,nzlcl
         read(iunit, end=10) rarr(x, y, z)
      enddo
   enddo
enddo
return
10 return
end

c-----------------------------------------------
c subroutine for writing input data (one local section at
c a time)
c-----------------------------------------------
subroutine writeproc(ignum, ounit, ipx, ipy, ipz, ixlo, ixhi,
- iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, warr)
implicit none
integer ignum, ounit
integer ipx, ipy, ipz, x, y, z
integer ixlo, ixyi, iylo, iyhi, izlo, izhi
integer nxlcl, nylcl, nzlcl
complex warr(ixlo:ixhi, iylo:iyhi, izlo:izhi)
do x=1,nxlcl
   do y=1,nylcl
      do z=1,nzlcl
         write(ounit) warr(x, y, z)
      enddo
   enddo
enddo
return
end

c-----------------------------------------------
c automatically-generated wrapper program to pack arrays
c and call set_mesh or redistribute_data
c-----------------------------------------------

include 'par_fft_wrap.h'

c-----------------------------------------------

Computational Benchmark Program.
c-----------------------------------------------
c example of mesh/spectral archetype
c
2D FFT

c benchmark version

c make-time parameters:
   NX, NY = dimensions of grid

c may need to adjust loops bounded by klim

-----------------------------------

-----------------------------------

main program

-----------------------------------

program main
 implicit none
 c include 'arch/_uparms/.h/
 c include 'arch/_parms/.h/
 c include 'bench/_fft/_header/.h/
 c processors (MAXPROCS)
 c call spawn(procs)
 c end

-----------------------------------

process-main program

-----------------------------------

subroutine procmain(
 implicit none
 c include 'arch/_uparms/.h/
 c include 'arch/_parms/.h/
 include 'bench/_fft/_header/.h/
 c integer nrepeats
 parameter (nrepeats=10)
 integer IUNIT, OUNIT
 parameter (IUNIT=11, OUNIT=12)
 character(*) FFT_IN
 parameter (FFT_IN='fft.IN')
 character(*) FFT_OUT1, FFT_OUT2
 parameter (FFT_OUT1='bench_fft', FFT_OUT2='OUT')
 c external readproc, writproc
 c integer procnum, procxyz(3), datasize(3)
 integer istarts, istarms, istops, istopms, itimediff, k, klim
 c arguments for cfft99:
 c cfft99(a, work, trigs, ifax, inc, jump, n, lot, isign)
 c a contains input/output of FFT (a set of vectors)
 c work is a work array
 c trigs, ifax contain constants for cfft99 (generated by
 c subroutine cftfax)
 c inc is stride between vector elements
 c jump is between successive vectors
 c n is size of each vector

87
c lot is how many vectors
isign indicates forward/inverse transform

integer ifaxnx(13), ifaxmy(13)
real trigsnx(2*nx_dgl), trigsny(2*ny_dgl)
integer inc, jump, n, lot

local section of distributed grid -- type is really complex,
declared double precision for compatibility with archetype
double precision arr(lclsize_dgl)
complex work(lclsize_dgl)

double precision arr(nx_dgl*ny_dgl)
complex work(nx_dgl*ny_dgl)

character*80 fft_out
ccharact*4 cbuff
integer cblen1

c call local_pos(procxyz, procnum, -1)
c if (procnum.eq.IOPROC) then
print*, 'procs = ', procnum
print*, 'nx_dgl = ', nx_dgl
print*, 'ny_dgl = ', ny_dgl
print*, 'nrepeats = ', nrepeats
endif
do m = 1, nrepeats
c------benchmark initializations
c set up grid and read input data
c
call mytime(istarts, istartms)
klim = 10
do k=1, klim
c if (procnum.eq.IOPROC) then
   open(unit=IUNIT, file=FFT_IN, form='unformatted')
endif

call read_mesh(readproc, id_dgl, IUNIT, arr)
call readproc(0, IUNIT, 1, 1, 1, nx_dgl, 1, ny_dgl, 1, 1, - nx_dgl, ny_dgl, 1, arr)
c if (procnum.eq.IOPROC) then
   close(IUNIT)
endif
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'reading input file: ', klim, itimediff,
  dble(itimediff)/dble(klim)
c

c initialize for fft
c
call mytime(istarts, istartms)
klim = 1000
do k=1, klim
call cftfax(nx_dgl, ifaxnx, trigsnx)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'initialization x: ', klim, itimediff,
  dble(itimediff)/dble(klim)
call mytime(istarts, istartms)
klim = 1000

88
do k=1,klim
   call cftfax(ny_dgl, ifaxn, trigsnx)
endo

call mytime(istarts, istartms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'initialization y: ', klim, itimediff,
$ dble(itimediff)/dble(klim)

------benchmark fft by rows

do fft by rows

call mytime(istarts, istartms)
klim = 50
do k=1,klim
   call local_pos(procxyz,procnum,id_dgl)
   call data_widths(procxyz,.false.,datasize,id_dgl)
   inc = datasize(1)
   jump = 1
   n = ny_dgl
   lot = datasize(1)
   lot = nx_dgl
   call cfft99(arr, work, trigsnx, ifaxnx, inc, jump, n, lot, -1)
endo
call mytime(istarts, istartms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'fft by rows: ', klim, itimediff,
$ dble(itimediff)/dble(klim)

------benchmark output

c write output data

------benchmark fft by columns

do fft by columns

call mytime(istarts, istartms)
klim = 50
do k=1,klim
   call local_pos(procxyz,procnum,id_dgl)
   call data_widths(procxyz,.false.,datasize,id_dgl)
   inc = nx_dgl
   jump = 1
   n = nx_dgl
   lot = datasize(2)
   lot = ny_dgl
   call cfft99(arr, work, trigsnx, ifaxnx, inc, jump, n, lot, -1)
endo
call mytime(istarts, istartms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'fft by column: ', klim, itimediff,
$ dble(itimediff)/dble(klim)

------benchmark output

c write output data

89
call mytime(istarts, istartms)
klim = 10
do k=1,klim
  c if (procnum.eq.IOPROC) then
  c build output filename (dummy version)
  write (cbuff, '(5i1/)') 0
  chlen1=l
  fft_out = FFT_OUT1 // FFT_OUT2
  open(unit=OUNIT, file=fft_out, form='unformatted')
  endif
  c call write_mesh(writeproc, id_dgl1, OUNIT, arr)
  call writeproc(0, OUNIT, 1,1,1,1,m_dgl1,1,ny_dgl1,1,1,
  mr_dgl1,ny_dgl1,1,arr)
  c if (procnum.eq.IOPROC) then
  close(OUNIT)
  endif
  enddo
  call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print/*, /'writing output file: ', klim, itimediff,
& dble(itimediff)/dble(klim)
c end of 'do m = 1, nrepeats' loop
enddo
  c return
end

subroutine readproc(ignum, iunit, ipx, ipy, ipz, ixlo, ixhi,
  - iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, rarr)
  implicit none
  integer ignum, iunit, x, y,z
  integer ipx, ipy, ipz
  integer ixlo, ixhi, iylo, iyhi, izlo, izhi
  integer nxlcl, nylcl, nzlcl
  complex rarr(ixlo:ixhi, iylo:iyhi, izlo:izhi)
  do x=1,nxlcl
    do y=1,nylcl
      do z=1,nzlcl
        read(iunit, end=10) rarr(x, y, z)
      enddo
    enddo
  enddo
10 return
end

subroutine writeproc(ignum, ounit, ipx, ipy, ipz, ixlo, ixhi,
  - iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, warr)
implicit none

integer ignum, ounit  
integer ipx, ipy, ipz, x, y, z  
integer ixlo, ixhi, iylo, iyhi, izlo, izhi  
integer nzlcl, nylcl, nzlcl

complex warr(ixlo:ixhi, iylo:iyhi, izlo:izhi)

do x=1,nxlcl  
do y=1,nylcl  
do z=1,nzlcl  
    write(ounit) warr(x, y, z)  
    enddo  
enddo  
enddo  
return  
end

/* automatically-generated wrapper program to pack arrays */
/* and call set_mesh or redistribute_data */

Communication Benchmark Program.

program main
implicit none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'arch_Parms.h'
include 'bench_mesh_spectral_header.h'

processors (MAXPROCS)
call Spawn(procs)
end

C------------------------------------------------------------------------
subroutine procmaint()
implicit none
include 'arch_uparms.h'
end
include './arch/_parms/.h/'
include './bench/_mesh/_spectral/_header/.h/'
c number of times to do communication operations
integer NTIMES
parameter (NTIMES=1)  
c sizes of data for broadcast operation
integer NIC, NRC, NDPC
parameter (NIC=2)
parameter (NRC=2)
parameter (NDPC=2)
integer IUNIT, OUNIT
parameter (IUNIT=11, OUNIT=12)
integer procs, procxyz (3), data size (3)
integer low (3), high (3)
integer glbstart (3), glbend (3)
integer lclstart (3), lclend (3)
integer locals (3), globals (3)
integer k, itemp1, itemp2, itemp3
logical iempty (3)
integer ioni (NIC)
integer ion (NRC)
integer domnum (NDPC)
double precision difflocal, diffmax
integer locmin, locmax
double precision data (lclsize_dg)
integer istarts, istartms, istops, istopms, itimediff, klim, k
external readproc, writeproc

call local_pos (procxyz, procnum, -1)
if (procnum .eq. DIPROC) then
  print*, 'procs = ', procs
  print*, 'data sizes:
  print*, 'nx_dg1 = ', nx_dg1
  print*, 'ny_dg1 = ', ny_dg1
  print*, 'broadcast sizes:
  print*, 'NIC = ', NIC
  print*, 'NRC = ', NRC
  print*, 'NDPC = ', NDPC
endif

C------ benchmark wrapper SETMESH

call mytime (istarts, istartms)
klim = 100000
  do k=1, klim
    call wrapper (id_dg1, id_blk, DOUBLEPRECISION, data, SETMESH)
  enddo

call mytimediff (istarts, istartms, istops, istopms, itimediff)
print*, 'SETMESH', procnum, ': ', klim, itimediff.

C------ benchmark local_pos

call mytime (istarts, istartms)
klim = 100000
  do k=1, klim
    call local_pos (procxyz, procnum, id_dg1)
  enddo

call mytimediff (istarts, istartms, istops, istopms, itimediff)
print*, 'local_pos', procnum, ': ', klim, itimediff,
$ \text{dble(itimediff)/dble(klim)}$

C---- benchmark data_widths

    call mytime(istarts, istartms)
    klim = 100000
    do k=1,klim
        call data_widths(procxyz, .false., datasize, id_dgl)
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms, itimediff)
    print*, 'data_widths ', procnum, ': ', klim, itimediff,
    $ \text{dble(itimediff)/dble(klim)}$

C---- benchmark wrapper REDISTRIBUTEDATA (one to blk)

    call mytime(istarts, istartms)
    klim = NTIMES
    do k=1,klim
        call wrapper(id_dgl, id_one, DOUBLEPRECISION_T, data, SETMESH)
        call wrapper(id_dgl, id_blk, DOUBLEPRECISION_T, data,
        $ \text{REDISTRIBUTEDATA)}$
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms, itimediff)
    print*, 'SETMESH/REDISTRIBUTEDATA (one to blk) ',
    $ \text{procnum. ':' }, klim, itimediff, \text{dble(itimediff)/dble(klim)}$

C---- benchmark wrapper REDISTRIBUTEDATA (blk to one)

    call mytime(istarts, istartms)
    klim = NTIMES
    do k=1,klim
        call wrapper(id_dgl, id_blk, DOUBLEPRECISION_T, data, SETMESH)
        call wrapper(id_dgl, id_one, DOUBLEPRECISION_T, data,
        $ \text{REDISTRIBUTEDATA)}$
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms, itimediff)
    print*, 'SETMESH/REDISTRIBUTEDATA (blk to one) ',
    $ \text{procnum. ':' }, klim, itimediff, \text{dble(itimediff)/dble(klim)}$

C---- benchmark wrapper REDISTRIBUTEDATA (row to col)

    call mytime(istarts, istartms)
    klim = NTIMES
    do k=1,klim
        call wrapper(id_dgl, id_row, DOUBLEPRECISION_T, data, SETMESH)
        call wrapper(id_dgl, id_col, DOUBLEPRECISION_T, data,
        $ \text{REDISTRIBUTEDATA)}$
    enddo
    call mytime(istops, istopms)
    call mytimediff(istarts, istartms, istops, istopms, itimediff)
    print*, 'SETMESH/REDISTRIBUTEDATA (row to col) ',
    $ \text{procnum. ':' }, klim, itimediff, \text{dble(itimediff)/dble(klim)}$

C---- benchmark wrapper REDISTRIBUTEDATA (col to row)

    call mytime(istarts, istartms)
    klim = NTIMES
    do k=1,klim
        call wrapper(id_dgl, id_col, DOUBLEPRECISION_T, data, SETMESH)
        call wrapper(id_dgl, id_row, DOUBLEPRECISION_T, data,
        $ \text{REDISTRIBUTEDATA)}$
    enddo

93
call mytime/(istops, istopms/)
call mytimediff/(istarts, istartms, istops, istopms, itimediff/)
print*, 'SETMESH/REDISTRIBUTEDATA (col to row) ',
$ procnum, ': ', klim, itimediff, dble(itimediff)/dble(klim)

C---- benchmark broadcast

call mytime/(istarts, istartms/)
klim = NTIMES
 do k/=/1, klim
  call broadcast(NIC, NRC, NDPC, iconst, rconst, dconst, -1)
 enddo
call mytime/(istops, istopms/)
call mytimediff/(istarts, istartms, istops, istopms, itimediff/)
print*, 'broadcast ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark data_bounds

call wrapper(id_dg/1/, id_blk, DOUBLEPRECISION_T, data, SETMESH)
call local_pos(procyz, procnum, id_dg/1/)
call mytime/(istarts, istartms/)
klim = 1000000
 do k/=/1, klim
  call data_bounds(procyz, low, high, id_dg/1/)
 enddo
call mytime/(istops, istopms/)
call mytimediff/(istarts, istartms, istops, istopms, itimediff/)
print*, 'data_bounds ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark pack3

call mytime/(istarts, istartms/)
klim = 1000000
 do k/=/1, klim
  call pack3(2, 2, 1, glbstart)
 enddo
call mytime/(istops, istopms/)
call mytimediff/(istarts, istartms, istops, istopms, itimediff/)
print*, 'pack3 ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C------ benchmark unpack3

call mytime/(istarts, istartms/)
klim = 1000000
 do k/=/1, klim
  call unpack3(itemp_1, itemp_2, itemp_3, glbstart)
 enddo
call mytime/(istops, istopms/)
call mytimediff/(istarts, istartms, istops, istopms, itimediff/)
print*, 'unpack3 ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark intersect

call wrapper(id_dg/1, id_blk, DOUBLEPRECISION_T, data, SETMESH)
call local_pos(procyz, procnum, id_dg/1/)
call pack3(2, 2, 1, glbstart)
call pack3(nx_dg/1/1, ny_dg/1/1, 1, glbend)
call mytime/(istarts, istartms/)
klim = 1000000
 do k/=/1, klim
  call intersect(procyz, glbstart, glbend, lclstart, lclend,
C---- benchmark local_to_global

call wrapper(id_dgl, id_blk,DOUBLEPRECISION_T,data,SETMESH)
call local_pos(procxyz,procnum,id_dgl)
call pack(1,1,1,locals)
call mytime(istarts, istartms)
klim = 1000000
do k=1,klim
call local_to_global(procxyz,locals,globals,id_dgl)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'local_to_global ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark boundary_exchange

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
call boundary_exchange(data,id_dgl)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'boundary_exchange (no wrap) ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark global_max_dp

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
call global_max_dp(difflocal,locmaxin,diffmax,locmaxout,id_dgl)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'global_max_dp ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark read_mesh (no actual I/O -- benchmarked elsewhere)

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
call wrapper(id_dgl, id_row,DOUBLEPRECISION_T,data,SETMESH)
c if (procnum.eq.IOPROC) then
c open(unit=IUNIT, file='fft_in', form='unformatted')
c endif
c call read_mesh(readproc, id_dgl, IUNIT, data)
c if (procnum.eq.IOPROC) then
c close(unit=IUNIT)
c endif
cenddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'read_mesh (dummy) ', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)
C---- benchmark write_mesh (no actual I/O -- benchmarked elsewhere)

call mytime(istarts, istartms)
klim = NTIMES

do k=1,klim
    call wrapper(id_dg1, id_row, DOUBLEPRECISION_T, data, SETMESH)
    if (procnum.eq.IOPROC) then
        open(unit=OUNIT, file=fft_out, form='unformatted')
        endif
    call write_mesh(writeproc, id_dg1, OUNIT, data)
    if (procnum.eq.IOPROC) then
        close(unit=OUNIT)
        endif
    enddo

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print/*, /'write_mesh (dummy)' , procnum, ': ' , klim, itimediff,
$ dble(itimediff)/dble(klim)
end

c------ dummy subroutine for reading input data (one local section at a time)

c------ subroutine readproc(ignum, iunit, ipx, ipy, ipz, ixlo, ixhi, iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, rarr)
      implicit none
      integer ignum, iunit, x, y, z
      integer ipx, ipy, ipz
      integer ixlo, ixhi, iylo, iyhi, izlo, izhi
      integer nzlcl, nylcl, nzlcl
      complex rarr(ixlo:ixhi, iylo:iyhi, izlo:izhi)

      do x=1,nxlcl
         do y=1,nylcl
            do z=1,nzlcl
               read(iunit, end=10) rarr(x, y, z)
            enddo
         enddo
      enddo
      return
      10 return
end

c------ dummy subroutine for writing input data (one local section at a time)

c------ subroutine writeproc(ignum, ounit, ipx, ipy, ipz, ixlo, ixhi, iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, warr)
      implicit none
      integer ignum, ounit
      integer ipx, ipy, ipz, x, y, z
      integer ixlo, ixhi, iylo, iyhi, izlo, izhi
integer nxlcl, nylcl, nzlcl
complex warr(ixlo:ixhi, iylo:iyhi, izlo:izhi)

  do x=1,nxlcl
    do y=1,nylcl
      do z=1,nzlcl
        write(ounit) warr(x, y, z)
      enddo
    enddo
  enddo
return
end

C---------------
include 'bench_mesh_spectral_wrap.h'

C---------------
B.2 Poisson Solver

Sequential Application Program.

  example of mesh/spectral archetype
  Poisson solver
  make-time parameters:
    NX, NY = dimensions of grid
  
C---------------

C---------------
main program
  
C---------------

program main
  implicit none
  include 'arch_uparms.h'
  include 'arch_parms.h'
  include 'seq_poisson_header.h'
  call procmain()
  end

C---------------

C---------------
process-main program

C---------------

c subroutine procmain()
  implicit none
  include 'arch_uparms.h'
  include 'arch_parms.h'
  include 'seq_poisson_header.h'
integer IUNIT, OUNIT
parameter (IUNIT=11, OUNIT=12)
character(*) P_IN
parameter (P_IN='poisson.IN')
character(*) P_OUT
parameter (P_OUT='seq_poisson.OUT')

c integer procnum, idummy(3)
c integer procnum_one, procxyz_one(3)
c integer procnum_blk, procxyz_blk(3)
c integer low(3), high(3)
c double precision uk(lclsize dg1)
c double precision ukp(1)(lclsize dg1)
c double precision F, G
external F, G
double precision diffmax
integer nsteps
call mytime(istarts, istartms, istops, istopms, itimediff

c read and broadcast constants
c
c call local_pos(idummy, procnum,-1)
c if (procnum .eq. IOPROC) then
  print*, 'procs = ', procs
  print*, 'nx dg1 = ', nx dg1
  print*, 'ny dg1 = ', ny dg1
  print*, 'nprocx blk = ', nprocx_blk
  print*, 'nprocy blk = ', nprocy_blk
  open(unit=IUNIT, file=P_IN, access='sequential',
$     status='old', form='formatted')
  read(IUNIT, *) H
  read(IUNIT, *) TOL
  read(IUNIT, *) NCHECK
  read(IUNIT, *) MAXSTEPS
  close(unit=IUNIT)
c endif
c call broadcast(NIC, NRC, NDPC, iconst, rconst, dconst,-1)
c
c set up grid (undistributed)
c
c call wrapper(id dg1, id_one, DOUBLEPRECISION_T, uk, SETMES)
c call local_pos(procxyz_one, procnum_one, id dg1)
c
c initialize undistributed grid
c
c if (procnum_one .eq. 1) then
c  call initgrid(nx dg1, ny dg1, uk)
c endif
c
c distribute
c
c call wrapper(id dg1, id blk, DOUBLEPRECISION_T, uk, REDISTRIBUTEDATA)
c call local_pos(procxyz_blk, procnum_blk, id dg1)
c
c perform computation
c call data_bounds(procxyz_blk, low, high, id_dgl)
c call computergrid(low(1), high(1), low(2), high(2), uk, ukp1, 
c - diffmax, nsteps)
c call computergrid(1, nx_dgl, 1, ny_dgl, uk, ukp1, diffmax, nsteps)
c redistribute to collect all data in one process again 
c call wrapper(id_dgl, id_one, DOUBLEPRECISION_T, uk, REDISTRIBUTEDATA)
c
print results
c if (procnum_one eq 1) then
open(unit=OUNIT, file=P_OUT, access='sequential', 
$ status='unknown', form='formatted')
write (OUNIT,*) 'NX, NY = ', nx_dgl, ny_dgl
write (OUNIT,*) 'H = ', H
write (OUNIT,*) 'tolerance = ', TOL
call Fprint(OUNIT)
call Gprint(OUNIT)
if (diffmax .le. TOL) then
write (OUNIT,*) 'convergence occurred in ', 
- nsteps, ' steps'
- print*, 'steps = ', nsteps
else
write (OUNIT,*) 'no convergence in ', MAXSTEPS,
- 'steps; ', 'max. difference ', diffmax
- print*, 'steps = ', MAXSTEPS
endif
call printgrid(nx_dgl, ny_dgl, uk, OUNIT)
close(unit=OUNIT)
c endif
c all mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'sequential time: ', itimediff
c return
cend

------------------------------------------------------------------------
c initialize grid
c------------------------------------------------------------------------

subroutine initgrid(nx, ny, uk)
imPLICIT none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'seq_poisson_header.h'
integer nx, ny
double precision uk(nx, ny)
double precision F, G
external F, G
integer i, j
c
interior points

do i = 2, nx-1

99
do j = 2, ny-1
   uk(i,j) = F(i,j,mx,ny,H)
enddo
c boundary points
c do j = 1, ny
   uk(1,j) = G(1,j,mx,ny,H)
   uk(nx,j) = G(nx,j,mx,ny,H)
doo do i = 2, mx-1
   uk(i,1) = G(i,1,mx,ny,H)
   uk(i,ny) = G(i,ny,mx,ny,H)
enddo
c /------------------
c print grid data
c subroutine printgrid(nx, ny, uk, ounit)
  implicit none
  include '/arch/_uparms/.h/'
  include '/arch/_parms/.h/
  include '/seq/_poisson/_header/.h/'
  integer nx, ny
  double precision uk(nx, ny)
  integer ounit
  integer i, j
do i = 1, nx
  if (nx .gt. 10) write (ounit,*), ' '
  write (ounit,9999) (uk(i,j), j = 1, ny)
enddo
9999 format(10F8.4)
do i = 1, nx, mx/5
   write (ounit,9999) (uk(i,j), j = 1, ny, ny/5), uk(i,ny)
enddo
write (ounit,9999) (uk(nx,j), j = 1, ny, ny/5), uk(nx,ny)
end

subroutine computegrid(ixlo,ixhi,iylo,iyhi,uk,ukp1,diffmax,nsteps)
  implicit none
  include '/arch/_uparms/.h/'
  include '/arch/_parms/.h+'
  include '/seq/_poisson_header.h/
  integer ixlo,ixhi,iylo,iyhi
  double precision uk(ixlo:ixhi,iylo:iyhi)
  double precision ukp1(ixlo:ixhi,iylo:iyhi)
  double precision diffmax
  integer nsteps

double precision F, G
external F, G
double precision diff, difflocal
c integer glbstart(3), glbend(3)
c integer lclstart(3), lclend(3)
c integer ipxyz(3), iproc
integer istart, iend, jstart, jend, kstart, kend
c integer locmaxin(3), locmaxout(3)
c logical iempty(3)
c integer i, j, k
c integer glbcl(3)
c integer istartg, jstartg, kstartg
c integer igoffset, jgoffset

c compute loop bounds
c call local_pos(ipxyz, iproc, id_dgl)
c call pack3(2, 2, 1, glbstart)
c call pack3(nx_dgl-1, ny_dgl-1, 1, glbend)
c call intersect(ipxyz, glbstart, glbend, lclstart, lclend, iempty, - id_dgl)
c call unpack3(istart, jstart, kstart, lclstart)
c call unpack3(ierr, jerr, kerr, lclend)
c compute starting global indices (of local section)
c call local_to_global(ipxyz, lclstart, glbcl, id_dgl)
c call unpack3(istartg, jstartg, kstartg, glbcl)
c compute offsets (global index = offset + local index)
c igoffset = istartg - istart
c jgoffset = jstartg - jstart

istart=2
jstart=2
kstart=1
iend=nx_dgl-1
jend=ny_dgl-1
kend=1
c igoffset=0
c jgoffset=0

c loop until convergence
diffmax = TOL + 1.0

do k = 1, MISTEPS

c refresh ghost boundaries

c call boundary_exchange(uk, id_dgl)

c compute new values:
do i = istart, iend
do j = jstart, jend
c !uko1(i, j) = 0.25*(H*H*F*(igoffset+i, jgoffset+j, -
  + uk(i-1, j) + uk(i, j-1)
  + uk(i+1, j) + uk(i, j+1)
  - uk(i-1, j) - uk(i, j-1)
  - uk(i+1, j) - uk(i, j+1)
enddo
c !eko1(i, j) = 0.25*(H*H*F*(i, j, nx_dgl, ny_dgl, -
  + uk(i-1, j) + uk(i, j-1)
  + uk(i+1, j) + uk(i, j+1)
enddo

c every MCHECK-th step, recompute convergence test
if (mod(k, MCHECK) .eq. 0) then
c difflocal = 0.0
diffmax = 0.0
do i = istart, iend
  do j = jstart, jend
    diff = abs(ukp(i,j) - uk(i,j))
    if (diff .gt. difflocal) difflocal = diff
    if (diff .gt. diffmax) diffmax = diff
  enddo
enddo

call global_max_dp(difflocal, locmaxin, diffmax, locmaxout, id_dg)
endif

copy new values to old values
do i = istart, iend
  do j = jstart, jend
    uk(i,j) = ukp(i,j)
  enddo
enddo

c check for convergence
nsteps = k
if (diffmax .le. TOL) go to 1000
enddo

1000 return
end

c---------------------------
c
functions for Poisson equation
c---------------------------

double precision function F(i,j, mx, ny, h)
implicit none
integer i, j, mx, ny
double precision h

  F = 0.0
end

subroutine Fprint(ounit)
integer ounit
implicit none
write (ounit,*)'F(i,j) = 0.0'
end

subroutine Gprint(ounit)
integer ounit
implicit none
write (ounit,*)'G(i,j) = (i+j)*h'
end

double precision function G(i,j, mx, ny, h)
implicit none
integer i, j, mx, ny
double precision h

  G = (i+j)*h
end

subroutine Gprint(ounit)
integer ounit
implicit none
write (ounit,*)'G(i,j) = (i+j)*h'
end

Parallel Application Program.

program main
    implicit none
    include 'arch_uparms.h'
    include 'archParms.h'
    include 'par_poisson_header.h'

    processors (MAXPROCS)
    call spawn (procs)
end

subroutine procmain()
    implicit none
    include 'arch_uparms.h'
    include 'archParms.h'
    include 'par_poisson_header.h'

    integer UNIT, UUNIT
    parameter (UNIT=11, UUNIT=12)
    character(*) P_IN
    parameter (P_IN='poisson.IN')
    character(*) P_OUT1, P_OUT2
    parameter (P_OUT1='par_poisson_IN', P_OUT2='OUT')

    integer procnum, idummy(3)

    end
integer procnum_one, procxyz_one(3)
integer procnum_blk, procxyz_blk(3)
integer low(3), high(3)
double precision uk(lclsize_dg1)
double precision ukp/lclsize_dg1)
double precision F, G
external F, G
double precision diffmax
integer nsteps
integer istarts, istartms, istops, istopms, itimediff

call mytime(istarts, istartms)
c
read and broadcast constants
c
call local_pos(idummy,procnum,-1)
if (procnum .eq. IOPROC) then
  open(unit=UNIT, file=P_IN, access='sequential',
$ status='old', form='formatted')
  read(UNIT, *) H
  close(UNIT)
print*, 'procs = ', procs
print*, 'nx_dg1 = ', nx_dg1
print*, 'ny_dg1 = ', ny_dg1
print*, 'nprocx_blk = ', nprocx_blk
print*, 'nprocy_blk = ', nprocy_blk
call broadcast(NIC,NRC,NDPC,iconst,rconst,dconst,-1)
c
set up grid (undistributed)
c
call wrapper(id_dg1, id_one, DOUBLEPRECISION_T, uk, SETMESH)
call local_pos(procxyz_one,procnum_one,id_dg1)
c
initialize undistributed grid
c
if (procnum_one .eq. 1) then
call initgrid(nx_dg1, ny_dg1, uk)
endif
c
distribute
c
call wrapper(id_dg1, id_blk, DOUBLEPRECISION_T, uk, REDISTRIBUTEDATA)
call local_pos(procxyz_blk,procnum_blk,id_dg1)
c
perform computation
c
call data_bounds(procxyz_blk, low, high, id_dg1)
call computegrid(low(1),high(1),low(2),high(2),ukp, - diffmax,nsteps)
c
redistribute to collect all data in one process again
call wrapper(id_dg1, id_one, DOUBLEPRECISION_T, uk, REDISTRIBUTEDATA)

print results

if (procnum_one .eq. 1) then
build output filename
if (nproc_blk .le. 9) then
write (cbuff1, '(3i1)') nproc_blk
chlen1=1
else
write (cbuff1, '(3i2)') nproc_blk
chlen1=2
endif
if (nproc_blk .le. 9) then
write (cbuff2, '(3i1)') nproc_blk
chlen2=1
else
write (cbuff2, '(3i2)') nproc_blk
chlen2=2
endif
p_out = P_OUT1 // cbuff1(1:chlen1) // '_' // cbuff2(1:chlen2)
- // P_OUT2
open(unit=OUNIT, file=p_out, access='sequential',
status='unknown', form='formatted')
write (OUNIT, 'N X, Y = ', 'm_dg1, ny_dg1
write (OUNIT, 'H = ', H
write (OUNIT, 'tolerance = ', TOL
call Fprint(OUNIT)
call Gprint(OUNIT)
if (diffmax .le. TOL) then
write (OUNIT, 'convergence occurred in ',
'nsteps, ' steps'
print*, 'steps = ', nsteps
else
write (OUNIT, 'no convergence in ', MAISTEPS,
'steps', ' max. difference ', diffmax
print*, 'steps = ', MAISTEPS
endif
call printgrid(nx_dg1, ny_dg1, uk, OUNIT)
close(unit=OUNIT)
endif

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'process ', procnum, ': ', itimediff
end

-----------------------------------------------
initialize grid
-----------------------------------------------

subroutine initgrid(nx, ny, uk)
imPLICIT none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'par_poisson_header.h'

integer nx, ny
double precision uk(nx, ny)
double precision F, G
external F, G
integer i, j

c
interior points

do i /= 2, nx/-1
  do j /= 2, ny/-1
    uk(i,j) = F(i,j,nx,ny,H)
  enddo
endo
c
boundary points

do j /= 1, ny
  uk(1,j) = G(1,j,nx,ny,H)
  uk(nx,j) = G(nx,j,nx,ny,H)
endo
do i /= 2, nx/-1
  uk(i,1) = G(i,1,nx,ny,H)
  uk(i,ny) = G(i,ny,nx,ny,H)
endo
c
print grid data

subroutine printgrid(nx, ny, uk, ounit)
implicit none
include '/arch/_uparms/.h/
include '/arch/_parms/.h/
include '/par/_poisson/_header/.h/
integer nx, ny
double precision uk(nx, ny)
integer ounit
integer i, j

do i /= 1, nx
  if (nx .gt. 10) write (ounit,*) ' '
  write (ounit,9999) (uk(i,j), j /= 1, ny)
endo
9999 format(10F8.4)
do i /= 1, nx, nx/5
  write (ounit,9999) (uk(i,j), j /= 1, ny, ny/5), uk(i,ny)
doendo
write (ounit,9999) (uk(nx,j), j /= 1, ny, ny/5), uk(nx,ny)
end

do grid computation
c
subroutine computegrid(ixlo,ishl,iylo,iyhi,uk,ukp1,diffmax,nsteps)
implicit none
include 'arch_uparms.h'
include 'arch_parms.h'
include 'par_poisson_header.h'

integer ixlo, ixhi, iylo, iyhi
double precision uk(ixlo:ixhi,iylo:iyhi)
double precision ukp(1(ixlo:ixhi,iylo:iyhi)
double precision diffmax
terger nsteps
double precision F, G
external F, G
double precision diff, difflocal
terger glbstart(3), glbend(3)
terger lclstart(3), lclend(3)
terger ipxyz(3), iproc
terger istart, iend, jstart, jend, kstart, kend
terger locmaxin(3), locmaxout(3)
terger empty(3)
terger i, j, k
terger glbcl(3)
terger istartg, jstartg, kstartg
terger igoffset, jegoffset

c compute loop bounds
  call local_pos(ipxyz,iproc,id_dg)
  call pack0(2,2,1,glbstart)
  call pack0(nz_dg=1, ny_dg=1, glbend)
  call intersect(ipxyz,glbstart,glbend,lclstart,lclend,empty, -
    id_dg))
  call unpack0((istart,jstart,kstart,lclstart)
  call unpack0((iend,jend,kend,lclend)

c compute starting global indices (of local section)
  call local_to_global(ipxyz,lclstart,glbclose,glbcl, id_dg)
  call unpack0((istartg,jstartg,kstartg,glbcl)

c compute offsets (global index = offset + local index)
  igoffset = istartg - istart
  jegoffset = jstartg - jstart

c loop until convergence
  diffmax = TOL + 1.0
  do k = 1,NISTEPS
    refresh ghost boundaries
    call boundary_exchange(uk,id_dg)
    compute new values
    do i = istart, iend
      do j = jstart, jend
        ukp(i,j) = 0.25*H*H*F*(igoffset+i,jgoffset+j, -
          * uk(i-1,j) + uk(i,j-1) -
          * uk(i+1,j) + uk(i,j+1) )
      enddo
    enddo
    every NCHECK-th step, recompute convergence test
    if (mod(k,NCHECK) .eq. 0) then
      difflocal = 0.0
      do i = istart, iend
        do j = jstart, jend
          diff = max(abs(ukp(i,j) - uk(i,j))
        endifdo
        if (diff .gt. difflocal) difflocal = diff
      enddo
enddo
   call global_max_dp(difflocal,locmaxin,diffmax,locmaxout,
   + id_dg)
endif

/!

// copy new values to old values
// do i = istart, iend
// do j = jstart, jend
// uk(i,j) = ukp1(i,j)
// enddo
// enddo

// check for convergence
// nsteps = k
// if (diffmax .le. TOL) go to 1000

// enddo
1000 return
end

====================================================================

// functions for Poisson equation
====================================================================

double precision function F(i,j,nx,ny,h)
implicit none
integer i, j, nx, ny
double precision h
F = 0.0
end

====================================================================

subroutine Fprint(ounit)
integer ounit
implicit none
write (ounit,*) 'F(i,j) = 0.0'
end

====================================================================

double precision function G(i,j,nx,ny,h)
implicit none
integer i, j, nx, ny
double precision h
G = (i+j)*h
end

====================================================================

subroutine Gprint(ounit)
integer ounit
implicit none
write (ounit,*) 'G(i,j) = (i+j)*H'
end

====================================================================

// automatically-generated wrapper program to pack arrays
// and call set_mesh or redistribute_data
//
include 'par_poisson_wrap.h'

Computational Benchmark Program.

main program

program main
  implicit none
  include 'arch_uparms.h'
  include 'arch_parms.h'
  include 'bench_poisson_header.h'
  call procmain()
end

subroutine procmain()
  implicit none
  include 'arch_uparms.h'
  include 'arch_parms.h'
  include 'bench_poisson_header.h'
  integer UNIT,UNIT
  parameter (UNIT=11,UNIT=12)
  character(*) P_IN
  parameter (P_IN='poisson.IN' )
  character(*) P_OUT1,P_OUT2
  parameter (P_OUT1='bench_poisson',P_OUT2=''.OUT')
  integer procmnum, idummy(3)
  integer procmnum_one, procryz_one(3)
  integer procmnum_blk, procryz_blk(3)
  integer low(3), high(3)
  double precision uk(1lclsize_dg1)
  double precision ukx1(lclsize_dg1)
  double precision uky1(ny_dg1)
double precision ukp(nx_dg1*ny_dg1+2)
double precision F, G
external F, G
double precision diffmax
integer nsteps
integer istarts, istartms, istops, istopms, itimediff, k, klim
character*80 p_out
character*4 cbuff1, cbuff2
integer cblen1, cblen2

c-----benchmark reading constants
 c read and broadcast constants
 c
c call local_pos(idummy,procnum,-1)
c if (procnum.eq. IOPROC) then
 c print*, 'procs = ', procs
 print*, 'nx_dg1 = ', nx_dg1
 print*, 'ny_dg1 = ', ny_dg1
 print*, 'nprocx_blk = ', nprocx_blk
 print*, 'nprocy_blk = ', nprocy_blk
 c
 call mytime(istarts, istartms)
klim = 100
do k=1,klim
 open(unit=IUNIT, file=P_IN, access='sequential',
 $ status='old', form='formatted')
 read(IUNIT, *) H
 read(IUNIT, *) TOL
 read(IUNIT, *) NCHECK
 read(IUNIT, *) MAXSTEPS
 close(unit=IUNIT)
 enddo
 call mytime(istops, istops)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
 print*, 'input/: ', klim, itimediff, dble(itimediff)/dble(klim)
c endif
 c call broadcast(NIC,NRC,NDPC,iconst,rconst,dconst,-1)
c
c set up grid (undistributed)
c
c call wrapper(id_dg1, id_one, DOUBLEPRECISION_T, uk, SETMES)
c call local_pos(procxyz_one, procnum_ow, id_dg1)

--------benchmark initialization (within initgrid)
c c initialize undistributed grid
c c if (procnum_one .eq. 1) then
 c call initgrid(nx_dg1, ny_dg1, uk)
c endif
c c distribute
c c call wrapper(id_dg1, id_blk, DOUBLEPRECISION_T, uk, REDISTRIBUTEDATA)
c call local_pos(procxyz_blk, procnum_blk, id_dg1)

--------benchmark computation (within computegrid)
c c perform computation
c c call data_bounds(procxyz_blk, low, high, id_dg1)
call computegrid(low(1), high(1), low(2), high(2), uk, ukp1,
diffmax, nsteps)
call computegrid(1, nx_dg1, 1, ny_dg1, uk, ukp1, diffmax, nsteps)
c redistribute to collect all data in one process again
c call wrapper(id_dg1, id_one, DOUBLEPRECISION_T, uk, REDISTRIBUTEDATA)
c print results
c if (procnum_one .eq. 1) then
call mytime(istarts, istartms)
klim = 100
k = 1, klim
do k = 1, klim
build output filename (dummy version)
write (cbuff1, '(3i1)') 0
chlen1 = 1
write (cbuff2, '(3i1)') 0
chlen2 = 1
p_out = P_OUT1 // P_OUT2
open(unit=OUNIT, file=p_out, access='sequential',
status='unknown', form='formatted')
write (OUNIT,*) 'NX, NY = ', nx_dg1, ny_dg1
write (OUNIT,*) 'H = ', TOL
write (OUNIT,*) 'tolerance = ', TOL
call Fprint(OUNIT)
call Gprint(OUNIT)
if (diffmax .le. TOL) then
write (OUNIT,*) 'convergence occurred in ', nsteps, ' steps'
c print, 'steps = ', nsteps
else
write (OUNIT,*) 'no convergence in ', MAXSTEPS, ' steps, ' 'max. difference ', diffmax
print, 'steps = ', MAXSTEPS
endif
call printgrid(nx_dg1, ny_dg1, uk, OUNIT)
close(unit=OUNIT)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print, 'output: ', klim, itimediff,
$ dble(itimediff)/dble(klim)
c endif
c return
c end

---
c initialize grid
c subroutine initgrid(nx, ny, uk)
implicit none
c include 'arch_uparms.h'
c include 'arch parms.h'
c include 'bench_poisson_header.h'
c include 'bench_poisson.h'
integer nx, ny
double precision uk(nx, ny)
double precision F, G
external F, G
integer i, j
integer istarts, istartms, istops, istopms, itimediff, k, klim

--- benchmark

call mytime(istarts, istartms)
klim = 1000
do k = 1, klim

c
interior points

do i = 2, nx-1
do j = 2, ny-1
   uk(i,j) = F(i,j,nx,ny,H)
endo
do
c
boundary points

do j = 1, ny
   uk(1,j) = G(1,j,nx,ny,H)
   uk(nx,j) = G(nx,j,nx,ny,H)
endo
do i = 2, nx-1
   uk(i,1) = G(i,1,nx,ny,H)
   uk(i,ny) = G(i,ny,nx,ny,H)
endo
do

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'initialization:', klim, itimediff,
$ dble(itimediff)/dble(klim)
end

--- print grid data

subroutine printgrid(nx, ny, uk, ounit)
   implicit none
   include 'arch_uparms.h'
   include 'arch_parms.h'
   include 'bench_poisson_header.h'

   integer nx, ny
double precision uk(nx, ny)
integer ounit
integer i, j

do i = 1, nx
   if (nx .gt. 10) write (ounit,*) ' '
   write (ounit,9999) (uk(i,j), j = 1, ny)
endo
9999 format(10F8.4)
do i = 1, nx, nx/5
   write (ounit,9999) (uk(i,j), j = 1, ny, ny/5), uk(i,ny)
endo
write (omit,9999) uk(nx,j), j = 1, ny, ny/5, uk(nx,ny)
end

c--do grid compuation
c

c subroutine computegrid(ixlo,ixhi,iylo,iyhi,uk,ukp/1/,diffmax,nsteps)
implicit none
c include 'arch_uparms.h'
c include 'arch_parms.h'
c include 'bench_poisson_header.h'
integer ixlo,ixhi,iylo,iyhi
double precision uk(ixlo:ixhi,iylo:iyhi)
double precision ukp/1/(ixlo:ixhi,iylo:iyhi)
double precision diffmax
integer nsteps
double precision F, G
external F, G
double precision diff, difflocal
c integer glbstart(/3/), glbend(/3/)
c integer lclstart(/3/), lclend(/3/)
c integer ipxyz(/3/), iproc
integer istart, iend, jstart, jend, kstart, kend
c integer locmaxin(/3/), locmaxout(/3/)
c logical iempty(/3/)
c integer igoffset, jgoffset
integer istartg, jstartg, kstartg
c integer igoffset, jgoffset
integer istarts, istarts, istops, istops, itimediff, k, klim

c compute loop bounds
c call local_pos(ipxyz,iproc,id dg)
c call pack3(2,2,1,glbstart)
c call pack3(mx dg-1, my dg-1,1,mbend)
c call intersect(ipxyz,gbstart,glbend,lclstart,lclend,iempty,
  - id dg)
c call unpack3(listart,jstart,kstart,lclstart)
c call unpack3(iend,jend,kend,lcend)
c compute starting global indices (of local section)
c call local_to_global(ipxyz,lclstart,gbcl1,1d dg)
c call unpack3(istartg,jstartg,kstartg,gbcl1)
c compute offsets (global index - offset + local index)
c igoffset = istartg - istart
c jgoffset = jstartg - jstart

istart=2
jstart=2
kstart=1
iend=mx dg-1
jend=my dg-1
kend=1
igoffset=0
jgoffset=0

c loop until convergence
diffmax = TOL + 1.0
c do k = 1, MAXSTEPS

 c refresh ghost boundaries
 call boundary_exchange(uk, id_dgl)

 c----- benchmark computation of new values
 c compute new values

 call mytime(istarts, istartms)
 klim = 1000
 do k = 1, klim

 do i = istart, iend
 do j = jstart, jend
 ukp1(i, j) = 0.25*(H*H*F*(igoffset*i, jgoffset*j, - nx_dgl, ny_dgl, H) - uk(i-1, j) + uk(i, j-1) - uk(i+1, j) + uk(i, j+1))
 enddo
 enddo

 call mytime(istops, istopms)
 call mytimediff(istarts, istartms, istops, istopms, itimediff)
 print*, 'computation: ', klim, itimediff,
 $ dble(itimediff)/dble(klim)

 c----- benchmark convergence test
 c every NCHECK-th step, recompute convergence test
 c if (mod(k, NCHECK) .eq. 0) then

 call mytime(istarts, istartms)
 klim = 1000
 do k = 1, klim

difflocal = 0.0
 do i = istart, iend
 do j = jstart, jend
 diff = abs(ukp1(i, j) - uk(i, j))
 if (diff .gt. difflocal) difflocal = diff
 enddo
 enddo

 call global_max_dp(difflocal, locmaxin, diffmax, locmaxout, id_dgl)
 enddo
 call mytime(istops, istopms)
 call mytimediff(istarts, istartms, istops, istopms, itimediff)
 print*, 'convergence test: ', klim, itimediff,
 $ dble(itimediff)/dble(klim)

 endif

 c----- benchmark copy
 c copy new values to old values

 call mytime(istarts, istartms)
 klim = 1000
 do k = 1, klim

 do i = istart, iend
 do j = jstart, jend
 uk(i, j) = ukp1(i, j)
 enddo

114
enddo

call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'copy: ', klim, itimediff,
$ 
dble(itimediff)/dble(klim)
c
check for convergence
c
nteps = k
c
if (diffmax .le. TOL) go to 1000
c
enddo

c1000 return
1000 continue
end

c---------------------------------------------------------------
c
c functions for Poisson equation
c
c---------------------------------------------------------------

double precision function F(i, j, nx, ny, h)
implicit none
integer i, j, nx, ny
double precision h

F = 0.0
end

c---------------------------------------------------------------

subroutine Fprint(ounit)
integer ounit
implicit none
write (ounit,*) 'F(i,j) = 0.0'
end

c---------------------------------------------------------------

double precision function G(i, j, nx, ny, h)
implicit none
integer i, j, nx, ny
double precision h

G = (i+j)*h
end

c---------------------------------------------------------------

subroutine Gprint(ounit)
integer ounit
implicit none
write (ounit,*) 'G(i,j) = (i+j)*h'
end

c---------------------------------------------------------------
c
c automatically-generated wrapper program to pack arrays
c and call set_mesh or redistribute_data
c---------------------------------------------------------------
Communication Benchmark Program.

program main
implicit none
include 'arch_uparms.h'
include 'archparms.h'
include 'bench_mesh_spectral_header.h'

processors (MAXPROCS)
call spawn(procs)
end

subroutine procmain(
implicit none
include 'arch_uparms.h'
include 'archparms.h'
include 'bench_mesh_spectral_header.h'

number of times to do communication operations
integer NTIMES
parameter (NTIMES=10)

sizes of data for broadcast operation
integer NIC, NRC, NDPC
parameter (NIC=2)
parameter (NRC=2)
parameter (NDPC=2)

integer UNIT,UNIT
parameter (UNIT=11,UNIT=12)

integer procnun, procxyz(3), datasize(3)
integer low(3), high(3)
integer globstart(3), globend(3)
integer lclstart(3), lclend(3)
integer locals(3), globals(3)
integer k, itemp1, itemp2, itemp3
logical empty(3)
integer ionst(NIC)
integer ronst(NRC)
integer donst(NDPC)

double precision difflocal, diffmax
integer locmaxin, locmaxout
double precision data(lclsize DG)
integer istarts, istartms, istops, istopms, itimediff, klim, k
external readproc, writeproc
call local_pos(procxyz, procnum, -1)
if (procnum .eq. UPNUM) then
  print*, 'procs = ', procs
  print*, 'data sizes:'
  print*, 'nx DG = ', nx DG
  print*, 'ny DG = ', ny DG
  print*, 'broadcast sizes:'
  print*, 'NIC = ', NIC
  print*, 'NRC = ', NRC
endif
C/-/-/-/- benchmark wrapper SETMESH
call mytime(istarts, istartms)
klim = 10000
do k=1, klim
call wrapper(id DG, id blk, DOUBLEPRECISION_T, data, SETMESH)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, itimediff)
print*, 'SETMESH ', procnum, ': ', klim, itimediff,
dble(itimediff)/dble(klim)
C---- benchmark local_pos
call mytime(istarts, istartms)
klim = 10000
do k=1, klim
call local_pos(procxyz, procnum, id DG)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, itimediff)
print*, 'local_pos ', procnum, ': ', klim, itimediff,
dble(itimediff)/dble(klim)
C---- benchmark data_widths
call mytime(istarts, istartms)
klim = 10000
do k=1, klim
call data_widths(procxyz, .false., datasize, id DG)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, itimediff)
print*, 'data_widths ', procnum, ': ', klim, itimediff,
dble(itimediff)/dble(klim)
C---- benchmark wrapper REDISTRIBUTEDATA (one to blk)
call mytime(istarts, istartms)
klim = NTIMES
do k=1, klim
call wrapper(id DG, id one, DOUBLEPRECISION_T, data, SETMESH)
call wrapper(id DG, id blk, DOUBLEPRECISION_T, data,
  REDISTRIBUTEDATA)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, itimediff)
print*, 'SETMESH/REDISTRIBUTEDATA (one to blk) ',
117
$ \text{procnum}, ': ', \text{klim}, \text{itimediff}, \text{dble(itimediff)/dble(klim)}$

C---- benchmark wrapper REDISTRIBUTEDATA (blk to one)

call mytime(istartx, istartms)
klim = NTIMES
do k=1, klim
call wrapper(id_dg1.id_blk.DDOUBLEPRECISION_T.data,SETMESH)
call wrapper(id_dg1.id_one.DDOUBLEPRECISION_T.data,
$ \text{REDISTRIBUTEDATA})$
enddo

call mytime(istopx, istopms)
call mytimediff(istartx, istartms, istopx, istopms, itimediff)
print*, 'SETMESH/REDISTRIBUTEDATA (blk to one)',
$ \text{procnum}, ': ', \text{klim}, \text{itimediff}, \text{dble(itimediff)/dble(klim)}$

c---- benchmark wrapper REDISTRIBUTEDATA (row to col)

call mytime(istartx, istartms)
klim = NTIMES
do k=1, klim
call wrapper(id_dg1.id_row.DDOUBLEPRECISION_T.data,SETMESH)
call wrapper(id_dg1.id_col.DDOUBLEPRECISION_T.data,
$ \text{REDISTRIBUTEDATA})$
enddo

call mytime(istopx, istopms)
call mytimediff(istartx, istartms, istopx, istopms, itimediff)
print*, 'SETMESH/REDISTRIBUTEDATA (row to col)',
$ \text{procnum}, ': ', \text{klim}, \text{itimediff}, \text{dble(itimediff)/dble(klim)}$

c---- benchmark wrapper REDISTRIBUTEDATA (col to row)

call mytime(istartx, istartms)
klim = NTIMES
do k=1, klim
call wrapper(id_dg1.id_col.DDOUBLEPRECISION_T.data,SETMESH)
call wrapper(id_dg1.id_row.DDOUBLEPRECISION_T.data,
$ \text{REDISTRIBUTEDATA})$
enddo

call mytime(istopx, istopms)
call mytimediff(istartx, istartms, istopx, istopms, itimediff)
print*, 'SETMESH/REDISTRIBUTEDATA (col to row)',
$ \text{procnum}, ': ', \text{klim}, \text{itimediff}, \text{dble(itimediff)/dble(klim)}$

c---- benchmark broadcast

call mytime(istartx, istartms)
klim = NTIMES
do k=1, klim
call broadcast(NIC,NRC,NDPC,iconst,rconst,dconst,-1)
enddo

call mytime(istopx, istopms)
call mytimediff(istartx, istartms, istopx, istopms, itimediff)
print*, 'broadcast ', procnum, ': ', \text{klim}, \text{itimediff},
$ \text{dble(itimediff)/dble(klim)}$

c---- benchmark data_bounds

call wrapper(id_dg1.id_blk.DDOUBLEPRECISION_T.data,SETMESH)
call local_pos(procyx.procnum, id_dg1)
call mytime(istartx, istartms)
klim = 100000
do k=1, klim
call data_bounds(procyx, low, high, id_dg1)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'data_bounds', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark pack3

call mytime(istarts, istartms)
klim = 1000000
do k=1,klim
    call pack3(2,2,1,glbstart)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'pack3', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark unpack3

call mytime(istarts, istartms)
klim = 1000000
do k=1,klim
    call unpack3(itemp1,itemp2,itemp3,glbstart)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'unpack3', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark intersect

call wrapper(id_dg1, id_blk, DOUBLEPRECISION_T, data, SETMESH)
call local_pos(procxyz,procnum,id_dg1)
call pack3(2,2,1,glbstart)
call pack3(nx_dg1-1,my_dg1-1,1,glbend)
call mytime(istarts, istartms)
klim = 1000000
do k=1,klim
    call intersect(procxyz,glbstart,glbend,lclstart,lclend,
    $ iempty,id_dg1)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'intersect', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark local_to_global

call wrapper(id_dg1, id_blk, DOUBLEPRECISION_T, data, SETMESH)
call local_pos(procxyz,procnum,id_dg1)
call pack3(1,1,1,locals)
call mytime(istarts, istartms)
klim = 1000000
do k=1,klim
    call local_to_global(procxyz,locals,globals,id_dg1)
enddo
call mytime(istops, istopms)
call mytimediff(istarts, istartms, istops, istopms, itimediff)
print*, 'local_to_global', procnum, ': ', klim, itimediff,
$ dble(itimediff)/dble(klim)

C---- benchmark boundary_exchange

call mytime(istarts, istartms)
klim = NTIMES
do k=1,klim
    call boundary_exchange (data, id_dgl)
enddo
call mytime (istops, istopms)
call mytimediff (istarts, istartms, istops, istopms, itimediff)
print*, 'boundary_exchange (no wrap)', procnum, ': ', klim, itimediff, dble (itimediff)/dble (klim)

C---- benchmark global_max_dp

call mytime (istarts, istartms)
klim = NTIMES
do k=1,klim
    call global_max_dp (difflocal, locmin, diffmax, locmaxout, id_dgl)
enddo
call mytime (istops, istopms)
call mytimediff (istarts, istartms, istops, istopms, itimediff)
print*, 'global_max_dp', procnum, ': ', klim, itimediff, dble (itimediff)/dble (klim)

C---- benchmark read_mesh (no actual I/O -- benchmarked elsewhere)

call mytime (istarts, istartms)
klim = NTIMES
do k=1,klim
    call wrapper (id_dgl, id_row, DOUBLEPRECISION_T, data, SETMESH)
    c if (procnum.eq.IOPROC) then
        open(unit=IUNIT, file='fft_in', form='unformatted')
    c endif
    call read_mesh (readproc, id_dgl, IUNIT, data)
    c if (procnum.eq.IOPROC) then
        close(unit=IUNIT)
    c endif
enddo
call mytime (istops, istopms)
call mytimediff (istarts, istartms, istops, istopms, itimediff)
print*, 'read_mesh (dummy)', procnum, ': ', klim, itimediff, dble (itimediff)/dble (klim)

C---- benchmark write_mesh (no actual I/O -- benchmarked elsewhere)

call mytime (istarts, istartms)
klim = NTIMES
do k=1,klim
    call wrapper (id_dgl, id_row, DOUBLEPRECISION_T, data, SETMESH)
    c if (procnum.eq.IOPROC) then
        open(unit=OUNIT, file='fft_out', form='unformatted')
    c endif
    call write_mesh (writeproc, id_dgl, OUNIT, data)
    c if (procnum.eq.IOPROC) then
        close(unit=OUNIT)
    c endif
enddo
call mytime (istops, istopms)
call mytimediff (istarts, istartms, istops, istopms, itimediff)
print*, 'write_mesh (dummy)', procnum, ': ', klim, itimediff, dble (itimediff)/dble (klim)
end

dummy subroutine for reading input data (one local section at
subroutine readproc(ignum, iunit, ipx, ipy, ipz, ixlo, ixhi, iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, rarr)

implicit none

integer ignum, iunit, x, y, z
integer ipx, ipy, ipz
integer ixlo, ixhi, iylo, iyhi, izlo, izhi
integer nxlcl, nylcl, nzlcl

complex rarr(ixlo:ixhi, iylo:iyhi, izlo:izhi)

do x = 1, nxlcl
   do y = 1, nylcl
      do z = 1, nzlcl
         read(iunit, end=10) rarr(x, y, z)
      enddo
   enddo
enddo
10 return
end

subroutine writeproc(ignum, ounit, ipx, ipy, ipz, ixlo, ixhi, iylo, iyhi, izlo, izhi, nxlcl, nylcl, nzlcl, warr)

implicit none

integer ignum, ounit
integer ipx, ipy, ipz, x, y, z
integer ixlo, ixhi, iylo, iyhi, izlo, izhi
integer nxlcl, nylcl, nzlcl

complex warr(ixlo:ixhi, iylo:iyhi, izlo:izhi)

do x = 1, nxlcl
   do y = 1, nylcl
      do z = 1, nzlcl
         write(ounit) warr(x, y, z)
      enddo
   enddo
enddo
10 return
end

C---------------------------------------------------------------------
C dummy subroutine for writing input data (one local section at a time)
C---------------------------------------------------------------------

include '/bench/_mesh/_spectral/_wrap.h'

C---------------------------------------------------------------------