Parallel 3D Adaptive Mesh Refinement in Titanium*

Geoff Pike† Luigi Semenzato‡ Phillip Colella‡ Paul N. Hilfinger†

Abstract

We describe a 3-dimensional adaptive mesh refinement Poisson solver. The complete program consists of about 3,500 lines of Titanium code and runs on both shared-memory and distributed-memory architectures. This paper focuses on the algorithm and on our experiences in writing AMR and tuning its performance.

1 Introduction

This paper is a case study in the use of an experimental programming language in implementing a useful numerical method—adaptive mesh refinement (AMR) for solving Poisson's equation, $\Delta\varphi = \rho$, over the cube $\Omega = [0,1]^3$. Poisson's equation and its close relatives arise in many applications such as fluid mechanics, gravitation, heat flow, and electromagnetics. Poisson solvers are also used as components of some other PDE solvers. The authors have been involved in the design and implementation of the Titanium language, a dialect of Java intended for use in parallel computation [9]. Here, we attempt to show that Titanium is well-suited to the implementation of an AMR Poisson solver.

Most practical techniques for solving Poisson's equation discretize the problem domain Ω into cells and take as input the value of ρ at each cell plus boundary conditions. The output is the approximate value of φ at each cell.

To use a finite difference method, with cell-centered values, on a uniform grid with mesh spacing h, we write

$$\phi_{\mathbf{i}} \approx \varphi((\mathbf{i} + \frac{1}{2}\mathbf{u})h) \;\;,\;\; \rho_{\mathbf{i}} = \rho((\mathbf{i} + \frac{1}{2}\mathbf{u})h) \quad ,$$

where $\mathbf{u} = (1,1,1)$. The Laplacian may then be approximated by

$$L(\phi)_{\mathbf{i}} = \frac{1}{h^2}(-6\phi_{\mathbf{i}} + \sum_{s=1}^{3}(\phi_{\mathbf{i}+\mathbf{e}^s} + \phi_{\mathbf{i}-\mathbf{e}^s})) \approx \Delta\varphi((\mathbf{i} + \frac{1}{2}\mathbf{u})h) \quad ,$$

where $\mathbf{e}^s$ is the unit vector in the s direction.

A naïve relaxation method might only communicate information between adjacent cells. *Multigrid relaxation* uses several levels of grids at different resolutions with each grid covering the entire problem domain. By communicating information both within a

*email: `{pike,hilfinger}@cs.berkeley.edu`, `luigi@semenzato.com`, `pcolella@lbl.gov`. This work was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense under contract F30602-95-C-0136, by the U.S. Department of Energy under contracts DE-AC03-76SF00098 (through the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences), and W-7405-ENG-48 (through Lawrence Livermore National Laboratory, subcontract No. B336568).

†EECS Computer Science Division, University of California at Berkeley

‡NERSC, Lawrence Berkeley National Laboratory

level and across levels, multigrid is able to converge in time linear in the number of cells (see, for example, Demmel's text [6]).

AMRPoisson is an extension of the multigrid algorithm for solving Poisson's equation on adaptively refined grids [8]. AMR allows grids at high levels of resolution to cover only a subset of the problem domain. The subset of interest for a given level is represented by a set of rectangular *patches*.

In our implementation of AMRPoisson, parallelism is mainly achieved by relaxing simultaneously on all patches at a given refinement level. For large enough patches, communication and synchronization overheads are small.

Titanium is a language and system for high-performance parallel scientific computing. Titanium uses Java as its base, thereby leveraging the advantages of that language and allowing us to focus attention on parallel-computing issues. The main additions to Java are immutable classes, multi-dimensional arrays, an explicitly parallel "single program multiple data" (SPMD) model of computation with a global address space, and region-based memory management (further described by Aiken and Gay [1]). For portability, the Titanium compiler outputs C code.

The main goal in the design of the Titanium language and compiler is performance. Although the compiler is actively under development, the performance of several Titanium programs is already satisfactory. In some cases, it exceeds the performance of hand-coded C or FORTRAN. Typical performance is 0.7x to 0.9x that of hand-coded C or FORTRAN. We will present our latest performance benchmarks at the conference.

A significant fraction of the AMRPoisson code is dedicated to computing boundary values at the interfaces between coarse and fine boundaries. This computation, although not onerous, is complex. Titanium's design aided us greatly in expressing this part of the calculation.

The next two sections describe the AMRPoisson algorithm and its implementation in Titanium. We then present performance results.

2 AMRPoisson

To extend multigrid to adaptively refined meshes, we need a multilevel discretization of Poisson's equation. First, we introduce some notation.

$$\begin{aligned}
l &= \text{a level number, } l \in \{0,\ldots,l_{max}\}; l_{max} \text{ is finest} \\
h^l &= \text{the mesh spacing for level } l \\
\Omega^l &= \text{a disjoint union of rectangular patches for level } l \\
P &= \text{the projection operator from } \Omega^{l+1} \text{ to } \Omega^l \\
R &= \text{the refining operator from } \Omega^l \text{ to } \Omega^{l+1}
\end{aligned}$$

The *refinement ratio* is the ratio $\frac{h^l}{h^{l+1}}$ and is typically a power of two. For simplicity we require that $R(P(\Omega^{l+1})) = \Omega^{l+1}$ and that cells at level l may only border cells at levels $l-1$, l, or $l+1$. Also, Ω^0 must cover the entire problem domain.

We let ϕ^l and ρ^l be cell-centered; they are defined on $\Omega^l - P(\Omega^{l+1})$. We are using a residual/correction formulation so we also need to store a residual r^l and a correction e^l; they are defined on all of Ω^l.

2.1 Discretization

We want a discretization of Poisson's equation of the form

$$L^l(\phi^l, \phi^{l-1}, \phi^{l+1}) = \rho^l \text{ on } \Omega^l - P(\Omega^{l+1}) \quad .$$

Away from the $l\,/\,l+1$ and $l\,/\,l-1$ boundaries, it should reduce to the uniform grid discretization defined above.

We compute L^l in two steps. First, we calculate $L^{NF,l}(\phi^l, \phi^{l-1})$ on Ω^l, which is an approximation to L^l computed under the assumption that we have information from the fine grid Ω^{l+1}. ("NF" stands for "no fine.") Away from the boundary with Ω^{l+1}, $L^{NF,l}$ is equal to L^l. Second, in an operation called *refluxing*, we correct the values of $L^{NF,l}$ at the cells adjacent to the $l\,/\,l+1$ boundary.

2.1.1 Computing $L^{NF,l}$ To compute $L^{NF,l}$, we want to reduce the problem to the solved case of a rectangle with uniform mesh spacing. We assume no information from the next finer level (if any), so we extend ϕ^l to $P(\Omega^{l+1})$ and fill it with some harmless value that will not cause roundoff problems. In the end we only care about L^l on $\Omega^l - P(\Omega^{l+1})$, and the later refluxing at the $l\,/\,l+1$ boundary is sufficient to erase any contribution made by the "harmless value" we have chosen.

We logically expand each rectangle in Ω^l (a disjoint union of rectangles) with a layer of *ghost cells*. Each ghost cell is filled with a value for ϕ thus:

- If the ghost cell is exterior to the problem domain, use the physical boundary conditions.
- If the ghost cell is covered by Ω^l, copy the value from ϕ^l.
- Otherwise, the ghost cell must be covered by Ω^{l-1}. We interpolate a value using a combination of ϕ^l and ϕ^{l-1} values. If one or more of the coarse cells required for interpolation is covered by a fine grid, then we must construct an appropriately accurate value for that coarse cell from the fine grid values.

We are now able to compute $L^{NF,l}$ on each rectangle of interest using the simple operator for uniform mesh spacing.

2.1.2 Refluxing $L^{NF,l}$ can be written in the following form:

$$L^{NF,l}(\phi^l, \phi^{l-1})_{\mathbf{i}} = \sum_{s=1}^{3} \frac{1}{h^l}(F^s_{\mathbf{i}+\frac{1}{2}\mathbf{e}^s} - F^s_{\mathbf{i}-\frac{1}{2}\mathbf{e}^s}) \quad ,$$

where the flux F^s is defined as

$$F^s_{\mathbf{i}+\frac{1}{2}\mathbf{e}^s} = \frac{1}{h}(\phi_{\mathbf{i}+\mathbf{e}^s} - \phi_{\mathbf{i}}) \quad .$$

L^l is computed by replacing the value of the flux on the $l\,/\,l+1$ boundary with that computed using the $l+1$ grid. For example,

$$L^l_{\mathbf{i}} = L^{NF,l}_{\mathbf{i}} - \frac{1}{h^l}(F^{s,l+1}_{\mathbf{i}+\frac{1}{2}\mathbf{e}^s} - F^{s,l}_{\mathbf{i}-\frac{1}{2}\mathbf{e}^s}) \quad .$$

So, as promised, L^l only depends on values of ϕ^l in $\Omega^l - P(\Omega^{l+1})$.

2.2 Extending Multigrid to AMR

The AMRPoisson algorithm is structured like a multigrid Poisson solve. We start at the finest level, repeatedly coarsen and relax to get a residual for the coarsest level, solve on the coarsest level, then repeatedly interpolate and relax until we are back to the finest level. (This is called a *V-cycle.*) The key difference is that AMRPoisson generalizes multigrid to the case where levels need not cover the entire problem domain.

The AMRPoisson algorithm relies on four sets of operators:

- The discretizations of Δ, i.e., $L^{NF,l}$ and L^l.
- A level smoothing operator $\mathrm{Smooth}(e, r)$, where e and r are defined on Ω^l.
- Averaging (A) and Interpolation (I) operators. The averaging operator maps quantities defined on Ω^k onto quantities defined on $P(\Omega^k)$. The interpolation operator maps quantities the other direction.
- A level 0 solver $\mathrm{Solve}(e, r)$ for solving the problem $L^0(e) = r$, where e and r are defined on Ω^0. We use one iteration of a vanilla multigrid solver.

The purpose of the level smoothing operator is to damp the high-frequency error. Operationally, $\mathrm{Smooth}(e^l, r^l)$ modifies its first argument by doing one iteration of Gauss-Seidel:

$$e^l_{\mathbf{i}} := e^l_{\mathbf{i}} + \lambda_{\mathbf{i}}(L^{NF,l}_{\mathbf{i}}(e^l, 0) - r^l_{\mathbf{i}})$$

with red-black ordering. The second argument to $L^{NF,l}$ is 0 because the smoothing operator is only working on one level. The relaxation parameter λ is set to $\frac{h^2}{4}$ for interior cells and $\frac{3h^2}{16}$ for boundary cells.

Figure 1 shows the complete AMRPoisson algorithm. We begin by setting the residual on the finest level, using physical boundary conditions and boundary conditions interpolated from the second finest level as necessary. Then, for each l from finest on down, we smooth the error at the current level, update phi, and set the residual for the next coarser level. Upon reaching level 0, we use our coarse level solver to improve ϕ^0. To complete the V-cycle we propagate improvements back up to the finest level. For each l from coarsest on up, we update the correction e^l by interpolating e^{l-1} on $P(\Omega^l)$, construct a separate correction δe^l that incorporates information from the rest of e^{l-1}, and update e^l and ϕ^l. The reason we cannot simply interpolate and then smooth the level is that the boundary condition for the smoothing would be incorrect, having not taken into account the coarser level correction on $\Omega^{l-1} - P(\Omega^l)$.

3 Implementation

In this section we try to give the flavor of Titanium and illustrate how useful it is for writing parallel scientific codes. Our goals in writing AMRPoisson were to write a small, clean program, but also to achieve good performance on symmetric multiprocessors (SMPs), distributed-memory supercomputers (e.g., Cray T3E, networks of workstations), and uniprocessors.

Titanium uses a *SPMD model* of parallelism; that is, a single program is concurrently executed by some number of processes, but the processes need not be in lockstep. Processes can synchronize and communicate using shared objects or arrays, or via language primitives such as broadcasts, barriers, scans, reductions, and locks. As the language stands, the number of processes is fixed at startup and there is no dynamic creation of threads as there is in Java.

AMRPoisson(ϕ, ρ)

$r^{lmax} := \rho^{lmax} - L^{NF,l}(\phi^{lmax}, \phi^{lmax-1})$

for $l = l_{max} \dots 1$

$\phi^{l,save} := \phi^l$

Smooth(e^l, r^l)

$\phi^l := \phi^l + e^l$

$e^{l-1} := 0$

$$r^{l-1} := \begin{cases} A(r^l - L^{NF,l}(e^l, e^{l-1})) & \text{on } P(\Omega^l) \\ \rho^{l-1} - L^{l-1}(\phi^{l-1}, \phi^{l-2}, \phi^l)) & \text{on } \Omega^{l-1} - P(\Omega^l) \end{cases}$$

end for

Solve(e^0, r^0)

$\phi^0 := \phi^0 + e^0$

for $l = 1 \dots l_{max}$

$e^l := e^l + I(e^{l-1})$

$r^l := r^l - L^{NF,l}(e^l, e^{l-1})$

$\delta e^l := 0$

Smooth($\delta e^l, r^l$)

$e^l := e^l + \delta e^l$

$\phi^l := \phi^{l,save} + e^l$

end for

end AMRPoisson

FIG. 1. *The AMRPoisson algorithm.*

```
static void pointRelax(double [3d] local fieldx,
                       double [3d] local rhsx,
                       double [3d] local factorx,
                       double hsquared,
                       Point<3> offset) {
  double [3d] local field = fieldx.translate(offset);
  double [3d] local rhs = rhsx.translate(offset);
  double [3d] local factor = factorx.translate(offset);
  foreach (p within rhs.domain() / [2, 2, 2]) {
    field[p * 2] += factor[p * 2] *
         (field[p * 2 + [ 0, 0, 1]] +
          field[p * 2 + [ 0, 0,-1]] +
          field[p * 2 + [ 0, 1, 0]] +
          field[p * 2 + [ 0,-1, 0]] +
          field[p * 2 + [ 1, 0, 0]] +
          field[p * 2 + [-1, 0, 0]] -
          6 * field[p * 2] - hsquared * rhs[p * 2]);
  }
}
```

FIG. 2. *A Titanium method for doing point relaxation. To allow red-black ordering, this method only computes on 1/8 of the points in* `rhs.domain()`*. One full pass with red-black ordering therefore requires 8 calls to* `pointRelax()`*.*

The Titanium language has a global address space, but not all objects are necessarily equally expensive to access. On networks of workstations, for example, code compiled by the Titanium compiler represents some references as $\langle proc, addr \rangle$ pairs. Access to a remote datum, which can occur by accessing an array element (`a[p]`) or a field of an object (`o.f`), generates communication. To save the run-time cost of checking whether a reference is remote, the programmer may statically specify it to be local by using the `local` type modifier. On SMPs and on uniprocessors the `local` modifier has no effect, and all pointers are represented simply as addresses.

Both the SPMD model and the distinction between local and not-necessarily-local references are also present in the Split-C [5] language. Split-C, FIDIL [7], and Java were three of the main influences on Titanium's design.

3.1 Example 1

More execution time is spent in the `pointRelax()` method than any other. See figure 2. The `pointRelax()` method is used in Gauss-Siedel relaxation (the Smooth(e^l, r^l) operator) and in computing $L^{NF,l}$. The only differences between this code and pure Java are the `foreach` control structure and the use of certain Titanium types. There are six examples of Titanium's multidimensional arrays, each 3-dimensional and local. As one might expect, we have arranged this most important method to operate on arrays known to be local. The `Point<3>` and `RectDomain<3>` types are also used. A `Point` is a tuple of integers. A `Domain` is a set of points. A `RectDomain` is a set of `Point`s that can be represented by an integer start, end, and stride for each dimension. The `foreach` construct simply iterates over a `Domain` or `RectDomain`—it is not a parallel construct. Parallelism is instead expressed with

```
foreach (p within AMR.processes.domain()) {
  AMRProcess ap = AMR.processes[p];
  foreach (ii within ap.levels[l].patches.domain()) {
    Patch otherPatch = ap.levels[l].patches[ii];
    if (this != otherPatch) {
      RectDomain<3> b = gdomain * otherPatch.domain;
      if (! b.isNull()) {
        RelatedPatch rp = new RelatedPatch();
        rp.domain = b;
        rp.patch = otherPatch;
        adjacentList.push(rp);
      }
    }
  }
}
adjacentPatches = adjacentList.toArray();
```

FIG. 3. *Calculation of* `adjacentPatches`*, an array of patches at the same level that border this patch.*

Titanium's SPMD model of computation.

Titanium arrays map indices from a `RectDomain`, i.e., `Points`, to some type. Various remapping operations are available, such as translation, reordering or scaling of dimensions, and so on.

Many Titanium programs, including AMRPoisson, alternate computation phases with communication phases. Before using the `pointRelax()` method, communication may be necessary to fill ghost regions. The SPMD model allows computation phases to proceed independently in each process; the number and size of patches may vary from process to process.

3.2 Example 2

One of the useful features of Titanium is the ability to directly manipulate `Points`, `Domains`, `RectDomains`. In boundary calculations, for example, it is useful to calculate a set of cells over which we shall iterate. Such sets can be easily represented in Titanium as `Domains` or `RectDomains`.

The example in figure 3 is taken from the `Patch` class in AMRPoisson. We consider each level Ω^l to be the union of n_l rectangular patches $\Omega^{l,k}$ for $k \in \{1, \dots, n_l\}$. Over the course of a V-cycle, computing values inside a patch often requires values from its neighboring patches at the same level, and from overlapping patches above and below. We pre-compute many of these relationships and reuse them several times. The figure shows code to compute adjacent patches at the same level. The outer loop iterates over `AMR.processes`, an array of per-process data. The inner loop iterates over each patch at level l "owned" by process p. Adjacency is determined by a brute force comparison of domains. If this patch (`this`) and another patch (`otherPatch`) are adjacent, then the domain of this patch extended by its ghost region (`gdomain`) and the domain of the other patch (`otherPatch.domain`) will have a non-empty intersection (`b`).

	1 processor	2 processors	4 processors	8 processors
time (sec)	151	78.9	43.4	28.9
speedup		1.91	3.47	5.22

FIG. 4. *Times are wall-clock times in seconds on a Sun Enterprise SMP, average of 3 runs, rounded to 3 significant figures. Each run performed 3 V-cycles on the grid hierarchy described in the text.*

4 Results and Discussion

Our implementation of AMRPoisson assumes that an adaptively refined grid hierarchy is provided as input. In addition, we assume that a distribution of grid patches to processes is part of the input. We are in the process of augmenting our implementation to compute a grid hierarchy and distribution using a variant of the methods used by Berger and Rigoutsos [3] and Berger and Colella [2].

We present results from a Sun Enterprise SMP with 8 processors and 2 gigabytes of RAM. Timing results are shown in figure 4. The grid hierarchy used for experiments has 5 levels as follows.

level	# of patches	size of patches
0	1	16x16x16
1	8	16x16x16
2	8	24x24x24
3	8	40x40x40
4	8	72x72x72

Complete results as of March, 1999, on SMPs and on the University of California's Network of Workstations (NOW), are available at `http://www.cs.berkeley.edu/projects/titanium/AMR/`. Also, we plan to compare our uniprocessor performance to an independent, sequential C++ implementation of the same algorithm now in progress.

In tuning the performance of parallel programs, we find it useful to visualize the bottlenecks with timelines. Traditional profiling tools provide little information on synchronization and load balancing. (see figure 5). The distribution of patches to processors is part of our input; timelines help to distinguish inherent inefficiencies due to load imbalances from inefficiencies due to performance bugs. Examining timelines allowed us to find several performance bugs in the code to do interlevel calculations. For example, in one case we were acquiring data from a finer patch one element at a time, with no adequate way to hide communication latency if the finer patch were expensive to access. A hand-inserted bulk prefetch was by far the more efficient option on distributed memory platforms.

5 Conclusion

Although both the Titanium language and compiler and the AMRPoisson application are still under development, we are pleased with our experience thus far. The algorithm's adaptivity is well-suited to Titanium's SPMD style; boundary calculations are easily expressed with the aid of the `Point` and `Domain` types; and Titanium arrays conveniently represent discretized grids. The ease of programming afforded by Titanium does not degrade performance, and may even be beneficial in so far as it makes programs easier to analyze and modify. Finally, relatively simple performance-measuring tools served us well in finding and fixing performance bottlenecks.

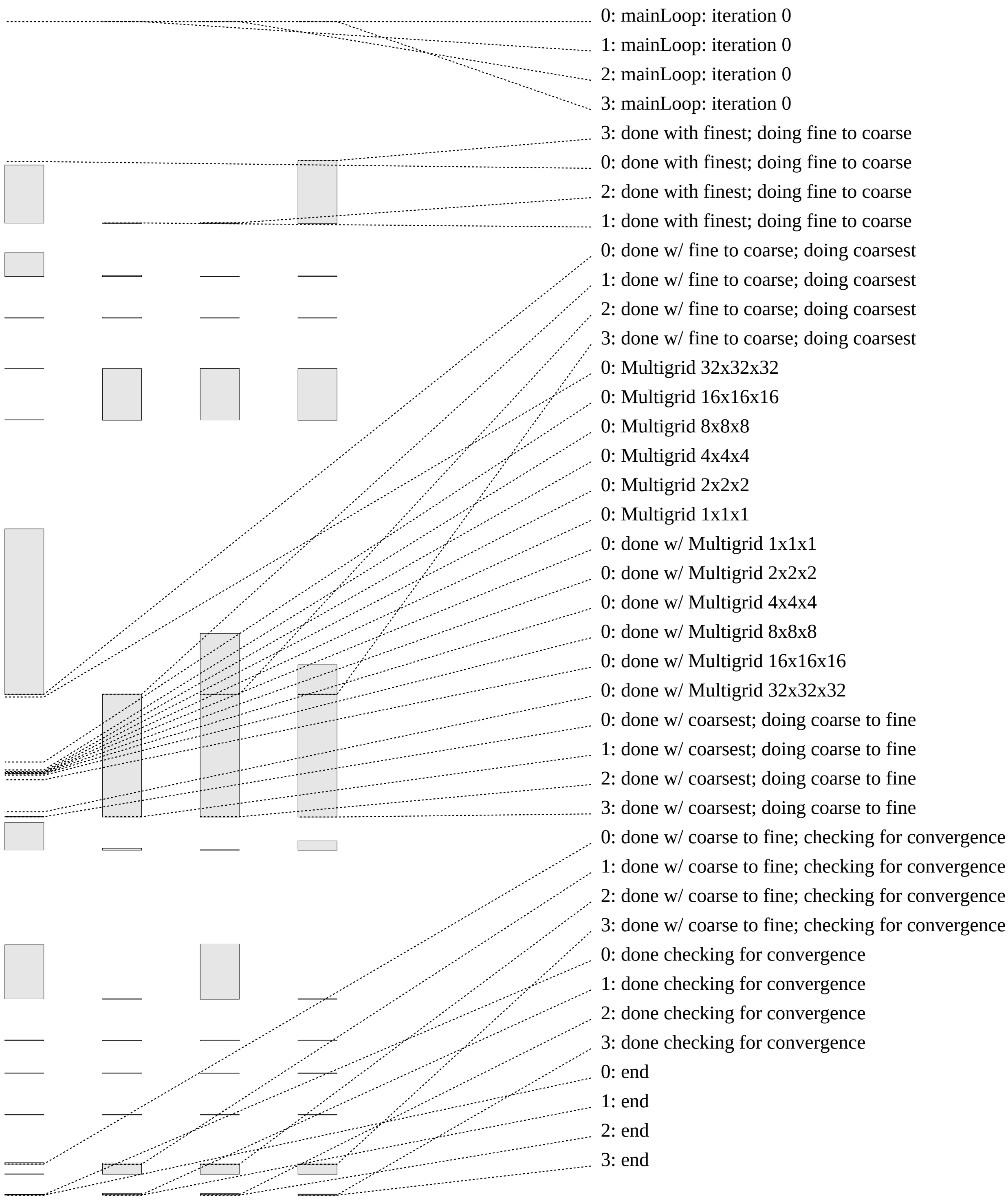

FIG. 5. *A timeline for a 4-processor run of AMRPoisson with a 32x32x32 grid at level 0. Time goes from top to bottom. At left is one column per processor. Any delay at a barrier is indicated by a grey box or, for insignificant delays, a solid horizontal line. At right, each event's label is preceded by a processor number.*

References

[1] A. Aiken and D. Gay. Memory Management with Explicit Regions. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[2] M. J. Berger and P. Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *J. of Comput. Phys.*, 82(1):64–84, May 1989.

[3] M. J. Berger and I. Rigoutsos. An Algorithm for Point Clustering and Grid Generation. *IEEE Transactions Systems, Man and Cybernetics*, 21(5):1278–1286, 1991.

[4] W. Y. Crutchfield and M. Welcome. Object Oriented implementation of adaptive mesh refinement algorithms. *Scientific Programming*, 2(4):145–156, 1993.

[5] D. E. Culler et al. Parallel Programming in Split-C. In *Supercomputing '93*, Portland, Oregon, November 1993.

[6] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[7] P. N. Hilfinger and P. Colella. FIDIL: A Language for Scientific Programming. In Robert Grossman, editor, *Symbolic Computing: Applications to Scientific Computing*, Frontiers in Applied Mathematics, chapter 5, pages 97–138. SIAM, 1989.

[8] D. Martin and K. Cartwright. Solving Poisson's equation using adaptive mesh refinement. Technical Report M96/66, University of California, Berkeley Electronic Research Laboratory, October 1996.

[9] K. Yelick et al. Titanium: A High-Performance Java Dialect. ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford, California, February 1998.