Performance Modeling of Iterative Solvers

1 An iterative solver algorithm

Iterative solvers are employed in a variety of numerical applications, and, as we will see later in the course, are a building block for more elaborate *multilevel* methods. For more information about this application, consult *Numerical Recipes in C* by Press et al.

More formally, we sill solve Poisson's equation in two dimensions:

$$\Delta u = f \text{ in } \Omega \tag{1}$$

for real-valued functions u and f of two variables. Ω is the computational box, a subset of R^2. We specify *Dirichlet boundary conditions* on $\partial\Omega$, the boundary of Ω:

$$u = g \text{ on } \partial\Omega \tag{2}$$

where g is also a real-valued function of two variables.

We discretize the computation using the method of finite differences; we now solve a set of discrete equations defined on `W`, a regularly-spaced $(N+2) \times (N+2)$ set of points in Z^2. We number the *interior* points of `W` from 1 to N in the x-coordinate and from 1 to N in the y-coordinate. The *boundary* points, which contain the Dirichlet boundary conditions for the problem, are located along x-coordinates of 0 and $N+1$ and along y-coordinates of 0 and $N+1$, as shown in Figure **??**.

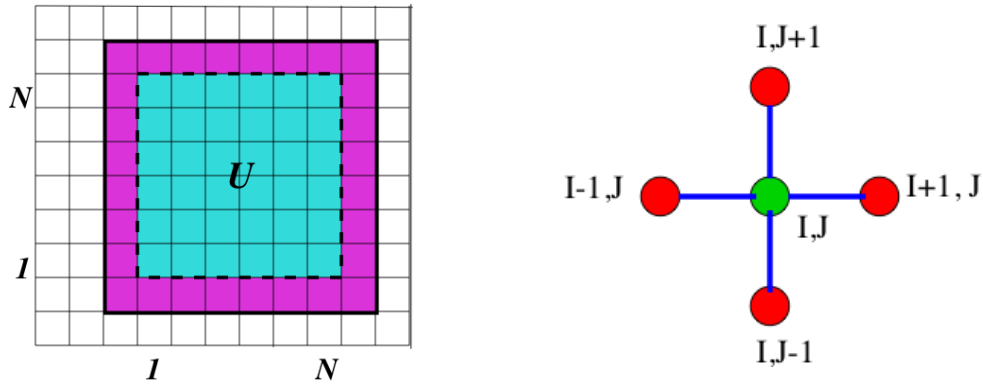

Figure 1: A finite difference mesh (left) and the accompanying finite difference stencil (right). The Grid `U` is defined over the 2-dimensional region with lower bound `(0,0)` and upper bound `(N+1,N+1)`. The solution is computed on the inner portion of the mesh; the boundary conditions are supplied on the outer edge. This boundary region is one cell thick, as dictated by the finite difference stencil for the smoother, which in our case updates each cell as a function of its nearest neighbors along the Manhattan directions.

When the right hand side function $f = 0$ in Eq. **??**, we have an important special case known as *Laplace's equation.* From now on we will consider this special case.

The simplest algorithm for solving Laplace's equation is known as *Jacobi's method*. Given an $(N+2) \times (N+2)$ mesh U, we compute a new mesh U^{new} such that to each value in the solution is the average of the 4 nearest neighbors in the "old" mesh. We then set the old mesh equal to the new one, and repeat the calculation until the *maximum* change in the solution over all the mesh values drops below some user-specified threshold ϵ, at which time we are finished[1] The algorithm is quite simple:

```
repeat
    forall (1 ≤ i, j ≤ N)
        u_{i,j}^{(s)} = (u_{i-1,j}^{(s-1)} + u_{i+1,j}^{(s-1)} + u_{i,j-1}^{(s-1)} + u_{i,j+1}^{(s-1)})/4
    endfor
until  u^{(s)} − u^{(s−1)}  small enough
```

where $u_{i,j}^{(s+1)}$ is the value of the mesh at (i, j) at iteration $s + 1$.

2 Factors Affecting Performance

Various factors will generally affect performance in a parallel implementation of the image smoothing algorithm

1. The rate at which we check for termination.
2. The data partitioning scheme
3. The ratio $\frac{N}{P}$.

We may improve performance of *computation* by checking for termination less frequently. However, we will reach a point of diminishing returns where the *running time* will start to increase, due to an increase in the number of iterations required to compute the answer to the desired accuracy. Remember: running time is the ultimate measure of performance!

The way we partition the data can also affect performance, along with the ratio of N to P. We'll derive a simple analytic performance model than can help us predict the effect of varying these two parameters, along with the partitioning strategy and the rate at which we check for termination. This model ignores some important effects, however, like the memory hierarchy (e.g. cache) and contention on the interprocessor communication, so it gives us only a rough estimate. We'll return to some of these effects later on.

3 Partitioning Model

We consider two types of partitioning, as shown in Fig. **??** and Fig. **??**: "strips," a one-dimensional scheme in which the cuts applied to the mesh extend completely across one extent of the mesh only, and "boxes," in which the cuts extend completely across both extents and the geometry of

[1]Other convergence criterion that may be more accurate, for example, the *residual*.

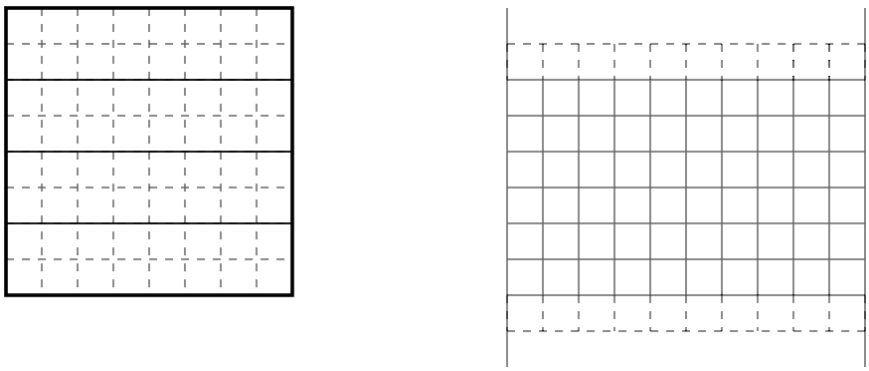

Figure 2: (a) Horizontal strip partitioning of the mesh showing (b) the location of the ghost cells. Vertical strip partitioning is not shown.

the boxes is square (We leave as an exercise the general case when the boxes are not square.) In the case of strips, the partitions may be either vertical or horizontal.

In choosing N and P we'll assume that work divides evenly among the processors, and that the extents of the local arrays are all greater than 2, and that there are no serial bottlenecks. For the two different types of partitionings we are considering, these assumptions imply that

- For strips: P divides N evenly, $\frac{N}{P} \geq 2$
- For boxes: $\sqrt{P}$ divides N evenly, $\frac{N}{\sqrt{P}} \geq 2$

4 The Performance Model

The analytic model predicts two timings:

- $T(1, N)$, the running time of the best serial algorithm,
- $T(P, N)$, the running time of the computation on P processors.

The first parameter used by our model is the *grind time.* The grind time is designated as $T_\gamma(P, N)$, and is the time taken to update a single point averaged over all updates. That is, we divide the running time by $N^2 \times N_{iter}$, where N_{iter} is the number of iterations:

$$T_\gamma(P, N) = \frac{T(P, N)}{N^2 \times N_{iter}} \tag{3}$$

Generally the grind time (and running time) is sensitive to P and N. In practice the compiler may render the grind time almost insensitive to these parameters, through a technique called *blocking for cache.* Let us designate $\gamma(N)$ as the grind time on a single processor, that is $T_\gamma(1, N)$. By definition,

$$T(1, N) = \gamma(N)N^2 N_{iter} \tag{4}$$

To treat the parallel case we need a more general form of the grind time which makes sense for problems that aren't square. This general form, $\gamma(m, n)$, gives the grind time for a problem of size

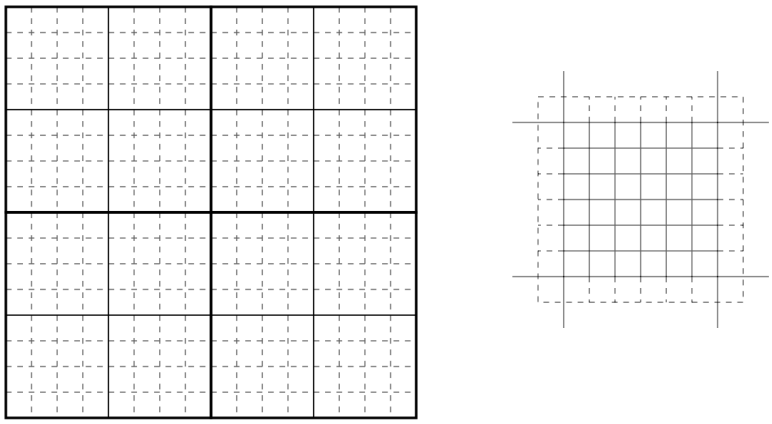

Figure 3: (a) Box partitioning of the mesh showing (b) the location of the ghost cells.

$m \times n$. (It follows that $\gamma(N) = \gamma(N, N)$.) In this case, the running time on a single processor is expressed as:

$$T(1, (m, n)) = \gamma(m, n)(mn)N_{iter} \tag{5}$$

With this definition in hand, we characterize T_P as follows:

$$T(P, N) = T(1, (m, n)) \ + \ T^{local}_{comm} + f * (T^{global}_{comm} + \kappa T(1, (m, n))) \tag{6}$$

where

- the $N \times N$ problem is divided into partitions of size $m \times n$;
- T^{local}_{comm} is the time spent communicating boundary data (the boundary regions are shown in Fig. **??** and Fig. **??**);
- termination is checked every f^{-1} iterations;
- T^{global}_{comm} is the communication cost in checking for termination; and
- κ is the scale factor relating the costs of the computation required to check for termination and for smoothing, e.g. if $\kappa = 0.6$, then the cost per point to compute the termination check is $\times 0.6$ the cost of doing a smooth.

In general

$$T(1, (m, n)) \ \leq \ \frac{T(1, N)}{P} \tag{7}$$

with equality holding when the compiler is able to handle memory locality appropriately. The above inequality implies that in partitioning the problem we are is likely to improve memory

locality. We can improve the running time on a single processor by simulating parallel execution, serially executing the smoother over the subsets of the problem corresponding to the partitions in the true parallel program.

5 Communication Costs

Let the message passing time be given by the relation

$$T_{message}(m) = \alpha + m\beta \tag{8}$$

where α is the *message startup time*, β is the *inverse peak message bandwidth*, and m is measured in *words*, and each pixel is a single word (double precision in our case). We may measure α and β using the `Ring` program supplied as part of assignment #1.[2]

The cost of the global communication is independent of whether we employ strip or block decomposition. It is simply

$$2\lceil log_2 P \rceil \alpha \tag{9}$$

The difference comes in local communication, which is used to fill the ghost cells prior to the start of each iteration.

5.1 Strip Decomposition

For the strip decomposition of Fig. **??**, each processor computes a chunk of $\frac{N}{P}$ complete rows of data. (We assumed that the mesh will be split evenly.) Thus, each processor sends and receives 2 sets of ghost cell data, coming from the two nearest neighboring processors. Each set of data contains N words. Thus, the local cost of communication for strips is:

$$T_{comm}^{local}(STRIPS) = 2(\alpha + \beta N) \tag{10}$$

5.2 Box Decomposition

For the box decomposition of Fig. **??**, each processor computes a square chunk of data $\frac{N}{\sqrt{P}}$ on a side (assuming that P is a perfect square, and $\sqrt{P}$ divides N evenly.) Each processor sends and receives four sets of ghost cell data, coming from the four nearest neighboring processors on the Manhattan directions (why aren't the corners needed?). Thus, the local cost of communication for boxes is:

$$T_{comm}^{local}(BOXES) = 4(\alpha + \beta \frac{N}{\sqrt{P}}) \tag{11}$$

6 Putting It All Together

To estimate the running time of our program, we determine α and β, and then measure $\gamma(m, n)$. To measure $\gamma(m, n)$ we must first determine the values of m and n. For strip partitioning we have two possibilities: $(m, n) = (1, N)$ or $(m, n) = (N, 1)$ depending on whether we have horizontal

[2] As we'll see later on, the `Ring` benchmark will predict an optimistic value for β. In practice, the actual peak bandwidth may be much lower. We will return to this later on.

or vertical strips. For box partitioning $m = n = \sqrt{p}$. We then run the best serial program on a problem of size $m \times n$ and use this timing to arrive at the appropriate grind time.

From our analysis we observe that communication with strips is $O(N)$ while communication with boxes is just $O(\frac{N}{\sqrt{P}})$. However, asymptotic analysis doesn't tell us the full picture since it ignores the the startup time α, which significantly affects performance. Let's ask the following question:

> Under what conditions will the strip decomposition lead to a shorter running time than the box decomposition?

To answer this question we need consider the local communication time only (Eq. **??** and Eq. **??**). A strip decomposition will lead to a faster running program when:

$$2(\alpha + \beta N) < 4(\alpha + \beta \frac{N}{\sqrt{P}}) \tag{12}$$

or (simplifying, assuming $P \geq 2$):

$$N < \frac{\sqrt{P}}{\sqrt{P} - 2} \frac{\alpha}{\beta} \tag{13}$$

For example, on $P = 16$ processors of the T3E, we require

$$N < \frac{2\alpha}{\beta} \tag{14}$$

But $\frac{\alpha}{\beta} = \frac{20\mu s}{0.004\mu s} = 5000$ and thus strips are faster when $N < 10000$. In practice it is unlikely that $N \geq 10000$ (we would want to use a more efficient solver like *multigrid*), and so we generally won't employ box decomposition contrary to what was predicted by asymptotic analysis.

7 Refinements to the Model

7.1 Memory strides

Earlier we mentioned that the value of β reported by the `Ring` program may be inaccurate. We now look at refinements to the our communication performance model that remove some of the innacuracies.

The value reported by Ring for peak inverse bandwidth β may actually be overstated by the Ring program in some cases. This can happen when the cost of sending a message also includes the cost of copying the bytes from the user's data structure into the network interface's memory buffers, or even into an intermediate user data structure. The cost of this copying depends on the *stride* of the memory copy, where the stride is the distance between adjacent elements that are transferred.

In the ring program, the stride is 1. However, in our parallel programs we may pass columns of a 2D array between processors, and these accesses have a stride of $N + 2$ (remembering that for a problem of size N we need an $(N + 2) \times (N + 2)$ mesh), the linear dimension of the mesh. If N is sufficiently large, that is a row of the mesh is larger than the cache line size, then successive

memory accesses along a column will not be able to exploit spatial locality in the cache. As a result, the bandwidth to memory will be reduced. Moreover, if we have to first pack the data into an intermediate user buffer, because we use a message passing primitive which assumes that data lies contiguously in memory, then the message passing costs increase still further.

In effect we need come up with three different values of β : β_x, β_y, and β_z, depending on whether the data to be transmitted come from a cut made along the x, y, or z axis, respectively. We may also need to add a term that is proportional to the number of cache lines actually touched. You will determine these values in your assignment.

7.2 Red Black Gauss-Seidel

A more efficient alternative to Jacobi's method is to employ Red/Black ordering. This algorithm divides the unknowns into two groups—red and black (sometimes called odd and even). This strategy decouples the points into disjoint sets, which may be updated in any desired order.

While there are various approaches such as successive over-relaxation (SOR), we will consider Gauss-Seidel's method. This method looks like Jacobi's method except that we use only one mesh. There is no need for a "new" mesh since the red and black updates are guaranteed to be independent within each sweep. The algorithm looks as follows:

```
repeat
    forall (1 ≤ i, j ≤ N, (i + j) is EVEN)
        u_{i,j} = (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})/4
    endfor
    forall (1 ≤ i, j ≤ N, (i + j) is ODD)
        u_{i,j} = (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})/4
    endfor
until δu small enough
```

When parallelizing this application we have the obvious optimization that we transmit only half the points when filling ghost cells. That is, we only transmit ODD or EVEN points. However, in the codes that you will be measuring, this optimization will not be implemented.

7.3 A smoother in 3 dimensions

We can apply the above techniques and analysis to a 3D iterative solver. In this case, each point receives the average of its *six* nearest neighbors: left, right, up, down, forward, behind. Various types of partitioning are possible including *slab* partitioning, in which slices are taken along one axis only, or 2D partitioning in which each processor gets a "stick" of work extending into the volume, or 3D partitioning in which each processor gets a mini-cube. The performance tradeoffs are more complicated and you will be asked to develop a model of performance for the 3D case.