SCA Numerical Assignment
Lab4: Parallelize a 3D Poisson problem

C. Álvarez

Fall 2024

Contents

Note: This laboratory is thought as an introduction to the final assignment. So, it is not optional and its mark will be the same as of the assignment itself. Also, you are supposed to spend more than the two hours allocated in the lab to be able to complete it.
As in the other labs, for the assignment you need to fill a report and deliver it in PDF at the "Racó". Also you must deliver the code that you have elaborated with your optimizations and any other file (like scripts) necessary to execute it in `boada` and obtain the results reported. Also, as in the previous labs, there are some **questions** on the document that will guide you to elaborate it. You should at least do all the tasks in the assignment part. If you are short of time don't go for the additional questions. They however, if correctly addressed, will help to improve your mark. You can do it by pairs.

Part 1

What to do in the Laboratory

In this laboratory you are going to deal with the final assignment. Although you can go for the assignment directly, it is useful that you spent some time dealing with the idiosyncrasies of the code in order to better tackle the difficulties of the assignment.

The purpose of this assignment is to introduce you to the ideas behind parallelizing iterative solvers for PDEs and related problems. Your goal is to parallelize a toy 3D Poisson problem solver based on preconditioned conjugate gradient iteration. The serial code is given; you should write an OpenMP or MPI version that (one hopes) gives reasonable speedup over several processors. Additionally you can try to address the additional questions.

1.1 The source files

You may start with the serial implementation supplied in Boada (**sessions/lab4.tar.gz**):

- **cgp3d.c**: code for the Laplace operator, preconditioners, and driver
- **pcg.c, pcg.h**: a serial PCG implementation
- **params.c, params.h**: solver parameters and command-line processing
- **timing.c, timing.h**: basic stopwatch timing utilities
- **Makefile**: a makefile
- **Makefile.in**: machine-specific settings used by the Makefile

1.2 Executing the code, the basics

First of all you should compile and execute the code submitting it to the queue system. It is very important that you remember to do all your measurements using the queue system in order to get consistent results. In order to understand the code you can look at it but it is also advisable that you read sections 2.2 and 2.7. Also you can try to execute with the `-h` parameter to get the builtin help.

Question 1: *Which is the default mesh size that the program is solving when you execute without parameters? How much time does it take? How many steps? Is the maximum number of steps enough to solve the problem correctly?*

Question 2: *How do you change the size of the mesh? Change it to 200 elements and answer again the previous question. How many steps do you need to correctly solve the problem now? How much time do you need?*

One of the first things that you have to take into account is that in order for the experiments to be useful you need to execute them always until the problem is correctly solved. An execution that finishes with an error larger than the default tolerance should not be taken into account (except to say that the problem can not be solved for this size/problem/configuration).

1.3 Executing the code, some more details

Now try to execute the code changing all the available parameters (mesh size, preconditioner, relaxation parameters and overlap). Note that not all the parameters affect all the executions.

Question 3: *Explain what are the different parameters and how they affect the problem. Which ones affect the problem regardless of the configuration? Which ones only apply when other parameters are used?*

Question 4: *Solve the problem with all the different preconditioners. How do the preconditioners affect the time to solution? And the necessary number of steps?*

Also, note that the right hand side can be changed although that it is done in the code.

Question 5: *How do you change the right hand side? How does it affect to the time to solution? Does the effect of the right hand side in the time to solution depend on the used preconditioner?*

1.4 Some hints for the optimization

Now that you have understood more or less the code, you can start doing the assignment itself. Here are some hints for the optimizations.

- In order to optimize the code remember that sometimes the best way to obtain performance from a parallel code is to parallelize the sequential version. Usually, conditionals are among the more performance damaging code sequences in a kernel.
- The schwarz code is, in fact, a blocking of the ssor code. However, the blocking can be changed.
- The ssor code presents a recursion that is similar to the one in the backward substitution problem.
- Copies can be parallelized and codes blocked.
- Remember to compute until the problem is solved.

Part 2

Numerical part assignment

2.1 Introduction

In this work we will be working on a solver for a discretization of Poisson's equation in three dimensions. In particular we are going to work on a three dimensional regular grid. For a set Ω in three dimensions (in our case, $\Omega = [0,1]^3$, Poisson's equation takes the form:

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f \quad \text{for} \quad (x,y,z) \in \text{int } \Omega$$

where int Ω is the interior of the set Ω and $\partial\Omega$ is the boundary of Ω. Here we will assume, $\Omega = [0,1]^3$. Let x_{ijk} for $i,j,k = 0,1,...,n$ be a set of mesh points. We can approximate the Laplacian of u (i.e. the sum of the second derivatives of u in the above equation), at a point by a finite difference method:

$$\begin{aligned}\left(-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2}\right)(x_{ijk}) \approx & \frac{-u(x_{i-1,jk}) + 2u(x_{ijk}) - u(x_{i+1,jk})}{h^2} + \\ & \frac{-u(x_{i,j-1,k}) + 2u(x_{ijk}) - u(x_{i,j+1,k})}{h^2} + \\ & \frac{-u(x_{ij,k-1}) + 2u(x_{ijk}) - u(x_{ij,k+1})}{h^2}\end{aligned}$$

where $h = 1/(n+1)$ is the mesh spacing. If we replace the Laplacian in the Poisson equation with this finite-difference approximation, we have a scheme for computing $u_{ijk} \approx u(x_{ijk})$:

$$\begin{aligned}6u(x_{ijk}) - u(x_{i-1,jk}) - u(x_{i+1,jk}) - u(x_{i,j-1,k}) - & \\ u(x_{i,j+1,k}) - u(x_{ij,k-1}) - u(x_{ij,k+1}) &= h^2 f_{ijk} \text{ for } 1 \le i,j,k \le n-1 \\ u_{pjk} &= 0 \text{ for } p \in \{0,n\} \\ u_{ipk} &= 0 \text{ for } p \in \{0,n\} \\ u_{ijp} &= 0 \text{ for } p \in \{0,n\}\end{aligned}$$

We can write this approximation as a matrix equation $Au = h^2 f$. Utilizing the matrix

$$T = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix}$$

A can be succinctly expressed as

$$A = I \otimes I \otimes T + I \otimes T \otimes I + T \otimes I \otimes I$$

where $X \otimes Y$ is the Kronecker product of matrices X and Y.

We have given you a serial code, which implements the conjugate gradients method for solving the above linear system. Furthermore, the method accepts a preconditioner. Conceptually, a preconditioner is a matrix that is applied to both sides of a linear system, which leads to a new system for which the method converges more quickly. In practice, the code has a function that applies the Laplacian operator that appears in the Poisson problem, and a second function that does an preconditioner solve. The preconditioner can be as simple as the identity (i.e. the code for it just copies the input to the output), but this will not speed things up. In the code we have given you, the preconditioner is one step of Symmetric Succesive OverRelaxation or SSOR, a method that could be used by itself iteratively to solve the problem. As posed, this preconditioner does not parallelize well; however, an additive Schwarz preconditioner built on top of it is more promising.

In the remainder of this document, we will sketch the basic (serial) code that you are to parallelize, give a list of tasks that you must complete, and suggest a few additional questions that you may wish to explore.

2.2 Solver parameters

The `solve_param_t` structure holds the parameters that describe the simulation. These parameters are filled in by the `get_params` function. Details of the parameters are described elsewhere in the code.

```
enum {              /* Types of preconditioners available: */
    PC_ID = 1,      /* 1. Identity */
    PC_SSOR = 2,    /* 2. SSOR */
    PC_SCHWARZ = 3 /* 3. Additive Schwarz */
};

typedef struct solve_param_t {
    int n;          /* Mesh size */
    int maxit;      /* Maximum PCG iterations */
    double rtol;    /* Relative residual convergence tolerance */
    int ptype;      /* Preconditioner type */
    double omega; /* SSOR relaxation parameter */
    int overlap;    /* Overlap size */
} solve_param_t;

int get_params(int argc, char** argv, solve_param_t* params);
```

2.3 Preconditioned CG

The PCG routine multiplies by A and M^{-1} through the `Mfun` and `Afun` function pointers (taking a vector length n, an opaque data object, an output buffer, and an input vector as arguments). We also pass `Mdata` and `Adata` as a way of getting context into the function calls[1]. In addition, we take storage for the solution (set to an initial guess on input) and the right hand side, as well as the maximum number of iterations allowed and a relative error tolerance.

The relative error tolerance is actually slightly subtle; we terminate the iteration when

$$\frac{\|r^{(k)}\|_{M^{-1}}}{\|r^{(0)}\|_{M^{-1}}} < \text{tol},$$

where $\|\cdot\|_{M^{-1}}$ refers to the norm induced by the M^{-1} inner product, i.e. $\|z\|^2_{M^{-1}} = zM^{-1}z$. This may or may not be the norm anyone actually cares about... but it surely is cheap to compute.

[1] This could admittedly be more convenient in C

```
double pcg(int n, mul_fun_t Mfun, void* Mdata,
           mul_fun_t Afun, void* Adata,
           double* restrict x,
           const double* restrict b,
           int maxit, double rtol)
{
   double* r = malloc(n*sizeof(double));
   double* z = malloc(n*sizeof(double));
   double* q = malloc(n*sizeof(double));
   double* p = malloc(n*sizeof(double));
   double rho0 = 0;
   double rho = 0;
   double rho_prev = 0;
   double rtol2 = rtol*rtol;
   int is_converged = 0;
   int step;

   tic(0);

   /* Form residual */
   Afun(n, Adata, r, x);
   for (int i = 0; i < n; ++i) r[i] = b[i]-r[i];

   for (step = 0; step < maxit && !is_converged; ++step) {
      Mfun(n, Mdata, z, r);
      rho_prev = rho;
      rho = dot(n, r, z);
      if (step == 0) {
         rho0 = rho;
         memcpy(p, z, n*sizeof(double));
      } else {
         double beta = rho/rho_prev;
         for (int i = 0; i < n; ++i) p[i] = z[i] + beta*p[i];
      }
      Afun(n, Adata, q, p);
      double alpha = rho/dot(n, p, q);
      for (int i = 0; i < n; ++i) x[i] += alpha*p[i];
      for (int i = 0; i < n; ++i) r[i] -= alpha*q[i];
      is_converged = (rho/rho0 < rtol2);
   }
   printf("%d steps, residual reduction %g (%s tol %g); time %g\n",
          step, sqrt(rho/rho0), is_converged ? "<=" : ">", rtol, toc(0));

   free(p);
   free(q);
   free(z);
   free(r);
   return rho/rho0;
}
```

2.4 3D Laplace operator

The 3D Laplacian looks like

$$(Ax)_{ijk} = h^{-1}\left(6x_{ijk} - \sum_{q,r,s:|q-i|+|j-r|+|k-s|=1} x_{qrs}\right)$$

The `mul_poisson3d` function applies the 3D Laplacian to an $n \times n \times n$ mesh of $[0,1]^3$ ($N = n^3, h = 1/(n-1)$), assuming Dirichlet boundary conditions.

```
void mul_poisson3d(int N, void* data,
                   double* restrict Ax,
                   double* restrict x)
{
   #define X(i,j,k) (x[((k)*n+(j))*n+(i)])
   #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])

   int n = *(int*) data;
   int inv_h2 = (n-1)*(n-1);
   for (int k = 0; k < n; ++k) {
      for (int j = 0; j < n; ++j) {
         for (int i = 0; i < n; ++i) {
            double xx = X(i,j,k);
            double xn = (i > 0) ? X(i-1,j,k) : 0;
            double xs = (i < n-1) ? X(i+1,j,k) : 0;
            double xe = (j > 0) ? X(i,j-1,k) : 0;
            double xw = (j < n-1) ? X(i,j+1,k) : 0;
            double xu = (k > 0) ? X(i,j,k-1) : 0;
            double xd = (k < n-1) ? X(i,j,k+1) : 0;
            AX(i,j,k) = (6*xx - xn - xs - xe - xw - xu - xd)*inv_h2;
         }
      }
   }

   #undef AX
   #undef X
}
```

2.5 Preconditioners for the Laplacian

2.5.1 The identity preconditioner

The simplest possible preconditioner is the identity (`pc_identity`):

```
void pc_identity(int n, void* data, double* Ax, double* x)
{
   memcpy(Ax, x, n*sizeof(double));
}
```

2.5.2 SSOR preconditioning

In terms of matrix splittings, if $A = L + D + L^T$ where D is diagonal and L is strictly lower triangular, the SSOR preconditioner is given by

$$M(\omega) = \frac{1}{2-\omega}(D/\omega + l)(D/\omega)^{-1}(D/\omega + L)^T,$$

where ω is a relaxation parameter. More algorithmically, SSOR means looping through the unknowns and updating each by adding ω times the Gauss-Seidel step; then doing the same thing, but with the

opposite order. Choosing an optimal value of ω is not all that easy; there are heuristics when SOR is being used as a stationary method, but I'm not sure that they apply when it is used as a preconditioner. The simplest thing to do is just to play with it. In order to apply SSOR preconditioning, we need the size n of the mesh (though in principle we could compute that from the number of mesh points) as well as the parameter ω. We store these parameters in a `pc_ssor_p3d_t` structure.

```
typedef struct pc_ssor_p3d_t {
   int n; /* Number of points in one direction on mesh */
   double omega; /* SSOR relaxation parameter */
} pc_ssor_p3d_t;
```

The `ssor_forward_sweep`, `ssor_backward_sweep`, and `ssor_diag_sweep` respectively apply $(D/\omega+L)^{-1}$, $(D/\omega+L)^{-T}$, and $(2-\omega)D/\omega$. Note that we're okay with ignoring the h^{-2} factor for computing the preconditioner - multiplying M by a scalar constant doesn't change the preconditioned Krylov subspace. Also note that these functions can operate on just part of the mesh (rather than the whole thing). This will be useful shortly when we discuss additive Schwarz preconditioners.

```
void ssor_forward_sweep(int n, int i1, int i2, int j1, int j2, int k1, int k2,
                        double* restrict Ax, double w)
{
   #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])
   for (int k = k1; k < k2; ++k) {
      for (int j = j1; j < j2; ++j) {
         for (int i = i1; i < i2; ++i) {
            double xx = AX(i,j,k);
            double xn = (i > 0) ? AX(i-1,j,k) : 0;
            double xe = (j > 0) ? AX(i,j-1,k) : 0;
            double xu = (k > 0) ? AX(i,j,k-1) : 0;
            AX(i,j,k) = (xx+xn+xe+xu)/6*w;
         }
      }
   }
   #undef AX
}

void ssor_backward_sweep(int n, int i1, int i2, int j1, int j2, int k1, int k2,
                         double* restrict Ax, double w)
{
   #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])
   for (int k = k2-1; k >= k1; --k) {
      for (int j = j2-1; j >= j1; --j) {
         for (int i = i2-1; i >= i1; --i) {
            double xx = AX(i,j,k);
            double xs = (i < n-1) ? AX(i+1,j,k) : 0;
            double xw = (j < n-1) ? AX(i,j+1,k) : 0;
            double xd = (k < n-1) ? AX(i,j,k+1) : 0;
            AX(i,j,k) = (xx+xs+xw+xd)/6*w;
         }
      }
   }
   #undef AX
}

void ssor_diag_sweep(int n, int i1, int i2, int j1, int j2, int k1, int k2,
                     double* restrict Ax, double w)
{
   #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])
   for (int k = k1; k < k2; ++k)
```

```
      for (int j = j1; j < j2; ++j)
         for (int i = i1; i < i2; ++i)
            AX(i,j,k) *= (6*(2-w)/w);
   #undef AX
}
```

Finally, the `pc_ssor_poisson3d` function actually applies the preconditioner.

```
void pc_ssor_poisson3d(int N, void* data,
                       double* restrict Ax,
                       double* restrict x)
{
   pc_ssor_p3d_t* ssor_data = (pc_ssor_p3d_t*) data;
   int n = ssor_data->n;
   double w = ssor_data->omega;

   memcpy(Ax, x, N*sizeof(double));
   ssor_forward_sweep(n, 0, n, 0, n, 0, n, Ax, w);
   ssor_diag_sweep(n, 0, n, 0, n, 0, n, Ax, w);
   ssor_backward_sweep(n, 0, n, 0, n, 0, n, Ax, w);
}
```

2.5.3 Additive Schwarz preconditioning

One way of thinking about Jacobi and Gauss-Seidel is as a sequence of local relaxation operations, each of which updates a variable (or a set of variables, in the case of block variants) assuming that the neighboring variables are known. With Jacobi and Gauss-Seidel, we update each variable exactly once in each pass. In Schwarz methods, we can update some variables with multiple relaxation operations in a single pass. To give an example, we will give an additive Schwarz (Jacobi-like) method that updates the bottom half of the domain and the top half of the domain with a little bit of overlap. To update the variables in each of the overlapping subdomains, we use one sweep of the SSOR step described in the previous section.

Schwarz-type preconditioners are a fantastic match for distributed memory computation, since the processors only communicate through the data in the overlap region. Note, though, that a specific variant (restrictive additive Schwarz) can't be used with conjugate gradient methods, because it does not yield symmetric preconditioners.

We describe the parameters for the Schwarz-type preconditioner with SSOR-based approximate solves in a `pc_schwarz_p3d_t` structure.

```
typedef struct pc_schwarz_p3d_t {
   int n; /* Number of mesh points on a side */
   int overlap; /* Number of points through the overlap region */
   double omega; /* SSOR relaxation parameter */
} pc_schwarz_p3d_t;
```

In order to compute independently on overlapping subdomains, we first get the local data from the vector to which we're applying the preconditioner; then we do an inexact solve on the local piece of the data; and then we write back the updates from the solve. The data motion is implemented in `schwarz_get` and `schwarz_add`.

```
void schwarz_get(int n, int i1, int i2, int j1, int j2, int k1, int k2,
                 double* restrict x_local, double* restrict x)
{
   #define X(i,j,k) (x[((k)*n+(j))*n+(i)])
   #define XL(i,j,k) (x_local[((k)*n+(j))*n+(i)])

   for (int k = k1; k < k2; ++k)
      for (int j = j1; j < j2; ++j)
```

```
        for (int i = i1; i < i2; ++i)
          XL(i,j,k) = X(i,j,k);

  if (k1 > 0)
    for (int j = j1; j < j2; ++j)
      for (int i = i1; i < i2; ++i)
        XL(i,j,k1-1) = 0;
  if (j1 > 0)
    for (int k = k1; k < k2; ++k)
      for (int i = i1; i < i2; ++i)
        XL(i,j1-1,k) = 0;
  if (i1 > 0)
    for (int k = k1; k < k2; ++k)
      for (int j = j1; j < j2; ++j)
        XL(i1-1,j,k) = 0;
  if (k2 < n-1)
    for (int j = j1; j < j2; ++j)
      for (int i = i1; i < i2; ++i)
        XL(i,j,k2+1) = 0;
  if (j2 < n-1)
    for (int k = k1; k < k2; ++k)
      for (int i = i1; i < i2; ++i)
        XL(i,j2+1,k) = 0;
  if (i2 < n-1)
    for (int k = k1; k < k2; ++k)
      for (int j = j1; j < j2; ++j)
        XL(i2+1,j,k) = 0;

  #undef XL
  #undef X
}

void schwarz_add(int n, int i1, int i2, int j1, int j2, int k1, int k2,
                 double* restrict Ax_local, double* restrict Ax)
{
  #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])
  #define AXL(i,j,k) (Ax_local[((k)*n+(j))*n+(i)])
  for (int k = k1; k < k2; ++k)
    for (int j = j1; j < j2; ++j)
      for (int i = i1; i < i2; ++i)
        AX(i,j,k) += AXL(i,j,k);
  #undef AXL
  #undef AX
}
```

The `pc_schwarz_poisson3d` function applies a preconditioner by combining independent SSOR updates for the bottom half plus an overlap region (a slap $n/2 + o/2$ nodes thick), then updating the top half plus an overlap region (another $n/2 + o/2$ node slab). The same idea could be applied to more regions, or to better approximate solvers.

```
void pc_schwarz_poisson3d(int N, void* data,
                          double* restrict Ax,
                          double* restrict x)
{
  double* scratch = malloc(N*sizeof(double));
  pc_schwarz_p3d_t* ssor_data = (pc_schwarz_p3d_t*) data;
  int n = ssor_data->n;
```

```
    int o = ssor_data->overlap/2;
    double w = ssor_data->omega;
    memset(Ax, 0, N*sizeof(double));

    schwarz_get (n, 0, n, 0, n, 0, n/2+o, scratch, x);
    ssor_forward_sweep (n, 0, n, 0, n, 0, n/2+o, scratch, w);
    ssor_diag_sweep (n, 0, n, 0, n, 0, n/2+o, scratch, w);
    ssor_backward_sweep(n, 0, n, 0, n, 0, n/2+o, scratch, w);
    schwarz_add (n, 0, n, 0, n, 0, n/2+o, scratch, Ax);

    schwarz_get (n, 0, n, 0, n, n/2-o, n, scratch, x);
    ssor_forward_sweep (n, 0, n, 0, n, n/2-o, n, scratch, w);
    ssor_diag_sweep (n, 0, n, 0, n, n/2-o, n, scratch, w);
    ssor_backward_sweep(n, 0, n, 0, n, n/2-o, n, scratch, w);
    schwarz_add (n, 0, n, 0, n, n/2-o, n, scratch, Ax);
}
```

2.6 Forcing functions

The convergence of CG depends not only on the operator and the preconditioner, but also on the right hand side. If the error is very high frequency, the convergence will appear relatively fast. Without a good preconditioner, it takes more iterations to correct a smooth error. In order to illustrate these behaviors, we provide two right-hand sides: a vector with one nonzero (computed via `setup_rhs0`) and a vector corresponding to a smooth product of quadratics in each coordinate direction (`setup_rhs1`).

```
void setup_rhs0(int n, double* b)
{
    int N = n*n*n;
    memset(b, 0, N*sizeof(double));
    b[0] = 1;
}

void setup_rhs1(int n, double* b)
{
    int N = n*n*n;
    memset(b, 0, N*sizeof(double));
    for (int i = 0; i < n; ++i) {
        double x = 1.0*(i+1)/(n+1);
        for (int j = 0; j < n; ++j) {
            double y = 1.0*(i+1)/(n+1);
            for (int k = 0; k < n; ++k) {
                double z = 1.0*(i+1)/(n+1);
                b[(k*n+j)*n+i] = x*(1-x) * y*(1-y) * z*(1-z);
            }
        }
    }
}
```

2.7 The `main` event

The `main` driver is pretty simple: read the problem and solver parameters using `get_params` and then run the preconditioned solve.

```
int main(int argc, char** argv)
{
```

```
    solve_param_t params;
    if (get_params(argc, argv, &params))
        return -1;

    int n = params.n;
    int N = n*n*n;

    double* b = malloc(N*sizeof(double));
    double* x = malloc(N*sizeof(double));
    double* r = malloc(N*sizeof(double));
    memset(b, 0, N*sizeof(double));
    memset(x, 0, N*sizeof(double));
    memset(r, 0, N*sizeof(double));

    /* Set up right hand side */
    setup_rhs1(n,b);

    /* Solve via PCG */
    int maxit = params.maxit;
    double rtol = params.rtol;

    if (params.ptype == PC_SCHWARZ) {
        pc_schwarz_p3d_t pcdata = {n, params.overlap, params.omega};
        pcg(N, pc_schwarz_poisson3d, &pcdata, mul_poisson3d, &n, x, b,
            maxit, rtol);
    } else if (params.ptype == PC_SSOR) {
        pc_ssor_p3d_t ssor_data = {n, params.omega};
        pcg(N, pc_ssor_poisson3d, &ssor_data, mul_poisson3d, &n, x, b,
            maxit, rtol);
    } else {
        pcg(N, pc_identity, NULL, mul_poisson3d, &n, x, b, maxit, rtol);
    }

    /* Check answer */
    mul_poisson3d(N, &n, r, x);
    double rnorm2 = 0;
    for (int i = 0; i < n; ++i) r[i] = b[i]-r[i];
    for (int i = 0; i < n; ++i) rnorm2 += r[i]*r[i];
    printf("rnorm = %g\n", sqrt(rnorm2));

    free(r);
    free(x);
    free(b);
}
```

2.8 Basic tasks

Your job is as follows:

- **Question 6:** *Time the unpreconditioned serial code on the Boada cluster. How much time does it take to solve different problem sizes (i.e. 100, 200 and 400)? How much time is spent in multiplying by A and doing dot products? Remember to specify the exact parameters that you are using for each execution.*

- **Question 7:** *Repeat your timing experiments with the preconditioned iteration. How much time is spent in the preconditioners per step? How many steps do the preconditioned iterations take? Remember to specify the exact parameters that you are using for each execution.*

- **Question 8:** *Parallelize the unpreconditioned code. You may use MPI or OpenMP; but if you use OpenMP you should think carefully about organizing your code to avoid contention and synchronization. Often the fastest programs in a shared memory model pretend that it's a shortcut for message passing! Be guided by what you find in your timing experiments - if you neglect to parallelize the most time-consuming parts of the code, you will not get good performance. After parallelizing the code, copy the modified code in the report and time it comparing to the original. Remember to specify the exact parameters that you are using for each execution.*

- **Question 9:** *Parallelize the Schwarz preconditioner. If you find yourself running short on time, at least make sure that you handle the degenerate case of block Jacobi (no overlap). After parallelizing the code, copy the modified code in the report and time it comparing to the original. Remember to specify the exact parameters that you are using for each execution.*

- **Question 10:** *Do scaling studies of the time required by your code for different values of n and p.*

2.9 Additional questions

Assuming time permits, take a crack at a couple of the following questions (that will reward you with additional mark):

- **Question 11:** ***(Additional)*** *Is there a more efficient way to code the multiplication by the Laplacian? What about the SSOR update?*

- **Question 12:** ***(Additional)*** *Have we chosen a good value of ω? How does the performance change as ω varies in the permissible range ($(0, 2)$)? Does the "right" value of ω change as n changes?*

- **Question 13:** ***(Additional)*** *How does the overlap parameter affect the convergence of our Schwarz iteration? What happens in the degenerate case of no overlap (block Jacobi)?*

- **Question 14:** ***(Additional)*** *How does the serial performance of PCG with SSOR compare to PCG with additive Schwarz? What about the parallel performance?*

- **Question 15:** ***(Additional)*** *How does the convergence behavior depend on the right hand side?*