Domen Problema

Veljko Petrović
Novembar, 2022

Zašto koristimo HPC?

Tipični problemi hpc-a

HPC kao priznanje neuspeha

- Na jedan veoma stvaran način, HPC je znak da smo se predali.
- Reformulacija problema nije radila.
- Lukavi algoritmi nisu radili.
- Sve što nam je ostalo jeste da napadnemo problem sa mnogo više resursa nego što je to normalno dostupno.
- HPC je, tako, zadnja linija odbrane kada se rešava nekakav problem.

HPC i ekonomija

- HPC je takođe ograničen neočekivanim ekonomskim faktorima.
- Vi možete lični računar namestiti da radi šta god želite: ako hoćete da uposlite 100% CPU-a generišući svetove kroz Dwarf Fortress niko se neće buniti. U najgorem slušaju trošite malo dodatne struje.
- HPC klasteri i super-računari su drugačiji: ozbiljne instalacije imaju troškove izvršavanja koji znače da se svaki minut mora naplaćivati.
- Stoga HPC se uglavnom koristi ili za projekte koje su interesantni vladama ili velikim kompanijama.

Neke popularne primene

- Kriptanaliza
- Naučna simulacija
- Simulacija fluida
- Simulacija plazme
- Simulacija klime
- Seizmološki modeli
- Statistička analiza ogromnih skupova podataka
- Problemi mašinskog učenja

Ograničenje kursa

- Ovde mi prelazimo ove primene samo ovlaš zato što, fundamentalno, svrha za koju se HPC primenjuje nije odgovornost HPC inženjera. To je odgovornost domenskog eksperta. HPC se bavi time da su resursi za proračun dostupni.
- Kao deo ovog kursa od vas će se tražiti da napišete seminarski rad o nekoj primeni zato što je zgodno da znate, makar okvirno, kako izgleda pravi problem ne bi li bili sposobniji da projektujete rešenja koja taj problem i njemu slične rešavaju.

Kriptanaliza

- Prvobitna primena računara (te i, po definicji, HPC-a).
- Cela svrha kriptografije jeste da ne ostavi nikakav brz način da se do podataka dođe bez nekog ključnog elementa (ključa, tipično).
- Kriptanaliza, idealno, razbije algoritam za enkripciju tako da to ne važi.
- Danas su kriptografi mnogo veštiji nego nekada i potpuno razbijeni algoritmi su retki. Alternativa jeste da se postigne parcijalno oslabljivanje gde se količina nagađanja i proračuna smanji donekle tako da (možda?) dođe u domet najbržih super-računara.
- Možete biti sigurni da negde u podrumima NSA zuje računari koji se bave baš ovim poslom.

Kriptanaliza—mogući problem

- Istražiti algoritam tipa opšteg sita polja brojeva (general number field sieve—GNFS) i koristiti ga za faktorisanje poluprostih brojeva.
- Poluprost (semiprime) je broj sa dva ne-trivijalna prosta faktora, npr. 2701 koji je 37 * 73.
- Ovo se onda lako može koristiti za razbijanje RSA enkripcije, tj. za ekstrakciju privatnog ključa iz javnog ključa.
- GNFS referentna implementacija je: https://sourceforge.net/projects/msieve/

Naučna simulacija

- Posmatrano sa veoma visokog nivoa apstrakcije, naučne simulacije su tipično (mada ne nužno) simulacije nekakvog fizičkog sistema koji je takav da je nemoguće dobiti rešenje zatvorene forme no je neophodno raditi aproksimaciju i diskretizaciju.
- Recimo u komputacionoj dinamici fluida prostor se deli na komade (ćelije) i specijalizovana verzija Navier-Stokes jednačina se rešava za svaki taj komad u svakom kvantu vremena formirajući rezultat.

Primer naučne simulacije—Problem n tela

- Problem n-tela je jedan od najpoznatijih i definitivno najstarijih problema u matematičkoj fizici.
- Imaju razne pod-formulacije koje sve objedinjuje sistem od n tela koji interaguju pod nekakvom silom no mi se ovde fokusiramo na vrlo jednostavnu, Njutnovsku verziju:
- Ako imamo n tela sa masama m0...n i pozicijama r0....n koje variraju u odnosu na vreme t, kako će ta tela da se ponašaju pod efektom gravitacije.
- (Dalji deo predavanja delimično baziran na materijalima Dr Rejmoda J. Spiterija za njegov kurs Parallel Programming for Scientific Computing držan 2014 u Univerzitetu Saskačevana, a koji su bazirani na materijalima sa Berklija.) https://www.cs.usask.ca/~spiteri/CMPT851/notes/nBody.pdf

Formulacija problema — ulazi

- U sistem ulaze:
  - Početne pozicije svih tela
  - Početne brzine svih tela
  - Mase svih tela
  - Vremenski parametri koji uključuju:
    - Koliko traje simulacija
    - Pod kojim koracima želimo izlaz

Formulacija problema—izlazli

- Iz sistema izlazi:
  - Pozicija svakog tela za vremenske korake
  - Brzina svakog tela za vremenske korake

Formulacija problema—pretpostavke

- Važe Njutnovi zakoni kretanja i gravitacije
- Svi objekti su tačkaste mase
- Svi objekti se mogu preklapati, tj. sudare modeliramo kao dva objekta na istom mestu.

Notacija

- Neka je n broj tela (čestica)
- Neka su mase m_i sa i od 0 do $n-1$.
- Neka su pozicije $\vec{r_i}(t)$ sa i od 0 do $n-1$, vektori.

Fizika

Onda je sila kojom čestica j deluje na česticu i iznosi:

$$\vec{f_{i,j}}(t) = -\frac{Gm_i m_j}{\|\vec{r_i}(t) - \vec{r_j}(t)\|^3}(\vec{r_i}(t) - \vec{r_j}(t))$$

Gde je:

$$G = 6.673 \times 10^{-11} m/(kg \cdot s^2)$$

Fizika

Ukupna sila na česticu će onda biti

$$\vec{F_i}(t) = \sum_{\substack{j=0 \\ i \neq i}}^{n-1} \vec{f_{ij}}(t)$$

Fizika

Što možemo da proširimo u:

$$\vec{f_{i,j}}(t) = \sum_{\substack{j=0 \\ i \neq i}}^{n-1} -\frac{Gm_i m_j}{\left\| \vec{r_i}(t) - \vec{r_j}(t) \right\|^3} (\vec{r_i}(t) - \vec{r_j}(t))$$

Fizika

A ovo se pojednostavljuje u:

$$\vec{f_{i,j}}(t) = -Gm_i \sum_{\substack{j=0 \\ i \neq i}}^{n-1} \frac{m_j}{\left\| \vec{r_i}(t) - \vec{r_j}(t) \right\|^3} (\vec{r_i}(t) - \vec{r_j}(t))$$

Fizika

Koristeći Njutnov drugi zakon u vektorskoj formulaciji imamo onda:

$$\vec{F_i}(t) = m_i \ddot{\vec{r_i}}(t), i = 0, 1, \dots n-1$$

Fizika

Ovo nas na kraju dovede do sistema prostih diferencijalnih jednačina drugog reda oblika:

$$\ddot{\vec{r_i}}(t) = -G \sum_{\substack{j=0 \\ i \neq i}}^{n-1} \frac{m_j}{\left\| \vec{r_i}(t) - \vec{r_j}(t) \right\|^3} (\vec{r_i}(t) - \vec{r_j}(t))$$

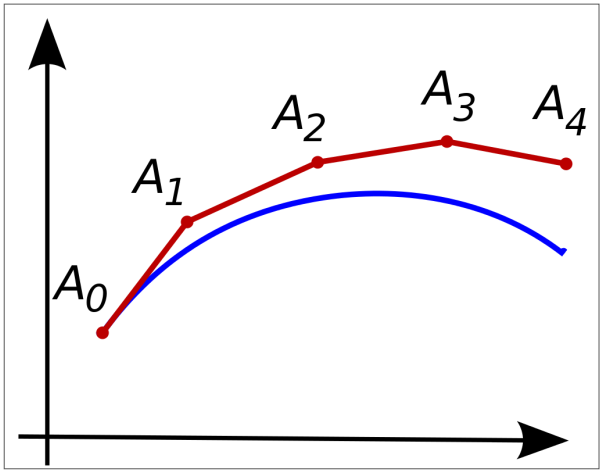
Neke olakšice

- Da bi nam život bio lakši, u ovoj ‘igračka’ implementaciji radimo sa sledećim ograničenjima:
  - Prostor je 2D i nije particionisan.
  - Jednačinu integralimo koristeći maksimalno jednostavnu Ojlerovu numeričku metodu koja gleda unapred.
  - Vremenski korak koji koristimo je konstantan.
- Ova ograničenja čine ovo vrlo lošim algoritmom za n-tela, ali vrlo ilustrativnim.
- Moguć zadatak: Istraživanje metoda za povećanje preciznosti i brzine modela n tela što uključuje hijerarhijsko particionisanje prostora, dinamičko zaokruživanje na nulu, i adaptivni korak vremena.

Ojlerova metoda?

- Prva i komično najprostija metoda za numeričku integraciju diferencijalnih jednačina.
- Poznata je i kao najprostija Runge-Kuta metoda.
- Esencijalno, aproksimiramo nepoznatu krivu datu kroz njen izvod tako što je podelimo na male deliće (koraci) u kojima se pretvaramo da je kriva linearna.

Ojlerova metoda?



Ojlerova metoda?

- Ovo nije osobito dobar način da se integrali diferencijalna jednačina.
- Može li bolje?
  - Nego šta.
  - Ali to je više posao za specijalistički kod, o kome više kasnije.

Gruba struktura algoritma

```
foreach timestamp{
    foreach particle p[i]{
        calculate F_i(t);
        update r_i(t);
        update v_i(t);
    }
}
return (r, v);
```

Gruba struktura proračuna Fi(t)

```
foreach particle j{
    if(j != i){
        dx = r[i].x - r[j].x;
        dy = r[i].y - r[j].y;
        d = sqrt(dx * dx + dy * dy);
        d3 = d*d*d;
        F[i].x -= G*m[i]*m[j]/d3*(r[i].x-r[j].x);
        F[i].y -= G*m[i]*m[j]/d3*(r[i].y-r[j].y);
    }
}
```

Malo ubrzanje

- Postoji fundamentalna simetrija u problemu kojom možemo da da prepolovimo broj proračuna
- Njutnov treći zakon znači da $f_{\vec{i,j}}(t) = -f_{\vec{j,i}}(t)$
- Ako znamo jednu, znamo i drugu, samo treba obrnuti znak.
- Kako da to uradimo lako?
- Najbolje je posmatrati matricu sile

Matrica sile

$$\begin{bmatrix} 0 & f_{0,1} & f_{0,2} & \cdots & f_{0,n-1} \\ -f_{0,1} & 0 & f_{1,2} & \cdots & f_{1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -f_{0,n-1} & -f_{1,n-1} & -f_{2,n-1} & \cdots & 0 \end{bmatrix}$$

Matrica sile

- Sad sve što treba da uradimo jeste da pravimo petlju samo nad gornjim/donjim trouglom matrice

Redukovani algoritam

```
foreach particle i{
    F_i(t) = 0;
}

foreach particle i{
    foreach particle j > i{
        F_i(t) += f_i,j(t);
        F_j(t) -= f_i,j(t);
    }
}
```

Dalji razvoj algoritma

Kada imamo ukupne sile koje deluju na neku česticu, onda možemo da kažemo:

$$a_i(t) = \ddot{r_i}(t) = \frac{F_i(t)}{m_i}$$

Ovo je naša diferencijalna jednačina koju hoćemo da integrišemo (dvaput) da bi dobili pomeraj tokom vremena.

Pojednostavljenje jednačine

Imamo Ojlerov metod koji smo pominjali ranije, ali on je za diferencijalne jednačine prvog reda oblika:

$\dot{y}(t) = f(t, y), y(0) = y_0, t > 0.$

Pojednostavljenje jednačine

- Mi imamo jednačinu drugog reda.
- Srećom, to se može popraviti: uvek je moguće pretvoriti jednačinu m-tog reda u m jednačina prvog reda tako što se funkcija i njeni izvodi do $m-1$ tretiraju kao nepoznate.

Pojednostavljenje jednačine

$$\dot{\vec{r_i}}(t) = \vec{v_i}(t)$$

$$\dot{\vec{v_i}}(t) = \frac{\vec{F_i}(t)}{m_i}$$

$$\vec{r_i}(0) = \vec{r_{i,0}}, \vec{v_i}(0) = \vec{v_{i,0}}$$

$$i = 0, 1, 2, \ldots, n-1$$

Ojlerov metod i pojedostavljena jednačina

```
r[i].x += dt*v[i].x;
r[i].y += dt*v[i].y;
v[i].x += dt*f[i].x/m[i];
v[i].y += dt*f[i].y/m[i];
```

Referentna implementacija rešenja problema n-tela

Osnovne implementacije

Uvodni komentari

- Slede par selekcija iz koda koji implementira rešenja problema n tela u različitim kontekstima
- Kod je adaptiran iz koda Univerziteta u Berkliju, i čiji je prvobitni autor Džejms Demel

Osnovna forma algoritma

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "timer.h"

#define DIM 2  /* Two-dimensional system */
#define X 0    /* x-coordinate subscript */
```

Osnovna forma algoritma

```
#define Y 1    /* y-coordinate subscript */

const double G = 6.673e-11;  /* Gravitational
        constant. */
                             /* Units are
        m^3/(kg*s^2)  */
// const double G = 0.1;  /* Gravitational
        constant. */
                       /* Units are m^3/(kg*s^2)
        */
```

Osnovna forma algoritma

```
typedef double vect_t[DIM];  /* Vector type for
        position, etc. */

struct particle_s {
   double m;  /* Mass     */
   vect_t s;  /* Position */
   vect_t v;  /* Velocity */
};
```

Osnovna forma algoritma

```
void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p);
void Get_init_cond(struct particle_s curr[], int
        n);
void Gen_init_cond(struct particle_s curr[], int
        n);
void Output_state(double time, struct particle_s
        curr[], int n);
```

Osnovna forma algoritma

```
void Compute_force(int part, vect_t forces[],
        struct particle_s curr[],
      int n);
void Update_part(int part, vect_t forces[], struct
        particle_s curr[],
      int n, double delta_t);
void Compute_energy(struct particle_s curr[], int
        n, double* kin_en_p,
      double* pot_en_p);
```

Osnovna forma algoritma

```
/*--------------------------------------------------
        --------------------*/
int main(int argc, char* argv[]) {
   int n;                    /* Number of
        particles        */
   int n_steps;              /* Number of
        timesteps        */
   int step;                 /* Current step
        */
   int part;                 /* Current particle
        */
   int output_freq;          /* Frequency of
        output        */
```

Osnovna forma algoritma

```
   double delta_t;           /* Size of timestep
        */
   double t;                 /* Current Time
        */
   struct particle_s* curr;  /* Current state of
        system    */
   vect_t* forces;           /* Forces on each
        particle    */
   char g_i;                 /*_G_en or _i_nput
        init conds */
#  ifdef COMPUTE_ENERGY
   double kinetic_energy, potential_energy;
```

Osnovna forma algoritma

```
#  endif
   double start, finish;       /* For timings
        */

   Get_args(argc, argv, &n, &n_steps, &delta_t,
        &output_freq, &g_i);
   curr = malloc(n*sizeof(struct particle_s));
   forces = malloc(n*sizeof(vect_t));
   if (g_i == 'i')
```

Osnovna forma algoritma

```
      Get_init_cond(curr, n);
   else
      Gen_init_cond(curr, n);

   GET_TIME(start);
#  ifdef COMPUTE_ENERGY
   Compute_energy(curr, n, &kinetic_energy,
        &potential_energy);
```

Osnovna forma algoritma

```
   printf("   PE = %e, KE = %e, Total Energy =
        %e\n",
         potential_energy, kinetic_energy,
        kinetic_energy+potential_energy);
#  endif
#  ifndef NO_OUTPUT
   Output_state(0, curr, n);
#  endif
   for (step = 1; step <= n_steps; step++) {
```

Osnovna forma algoritma

```
      t = step*delta_t;
//    memset(forces, 0, n*sizeof(vect_t));
      for (part = 0; part < n; part++)
         Compute_force(part, forces, curr, n);
      for (part = 0; part < n; part++)
         Update_part(part, forces, curr, n,
         delta_t);
#     ifdef COMPUTE_ENERGY
```

Osnovna forma algoritma

```
      Compute_energy(curr, n, &kinetic_energy,
        &potential_energy);
      printf("   PE = %e, KE = %e, Total Energy =
        %e\n",
          potential_energy, kinetic_energy,
        kinetic_energy+potential_energy);
#     endif
#     ifndef NO_OUTPUT
      if (step % output_freq == 0)
         Output_state(t, curr, n);
```

Osnovna forma algoritma

```
#     endif
   }

   GET_TIME(finish);
   printf("Elapsed time = %e seconds\n", finish-
        start);

   free(curr);
```

Osnovna forma algoritma

```
   free(forces);
   return 0;
}  /* main */


/*--------------------------------------------------
       ---------------------
 * Function: Usage
```

Osnovna forma algoritma

```
 * Purpose:  Print instructions for command-line
       and exit
 * In arg:
 *    prog_name:  the name of the program as typed
       on the command-line
 */
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of
       particles> <number of timesteps>\n",
        prog_name);
```

Osnovna forma algoritma

```
    fprintf(stderr, "   <size of timestep> <output
        frequency>\n");
    fprintf(stderr, "   <g|i>\n");
    fprintf(stderr, "   'g': program should
        generate init conds\n");
    fprintf(stderr, "   'i': program should get
        init conds from stdin\n");

    exit(0);
}   /* Usage */
```

Osnovna forma algoritma

```
/*--------------------------------------------------
        ---------------------
 * Function:  Get_args
 * Purpose:   Get command line args
 * In args:
 *    argc:            number of command line args
```

Osnovna forma algoritma

```
 *    argv:            command line args
 * Out args:
 *    n_p:             pointer to n, the number of
        particles
 *    n_steps_p:       pointer to n_steps, the
        number of timesteps
 *    delta_t_p:       pointer to delta_t, the
        size of each timestep
 *    output_freq_p:   pointer to output_freq,
        which is the number of
 *                     timesteps between steps
        whose output is printed
```

Osnovna forma algoritma

```
 *    g_i_p:           pointer to char which is
        'g' if the init conds
 *                     should be generated by the
        program and 'i' if
 *                     they should be read from
        stdin
 */
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p) {
   if (argc != 6) Usage(argv[0]);
```

Osnovna forma algoritma

```
    *n_p = strtol(argv[1], NULL, 10);
    *n_steps_p = strtol(argv[2], NULL, 10);
    *delta_t_p = strtod(argv[3], NULL);
    *output_freq_p = strtol(argv[4], NULL, 10);
    *g_i_p = argv[5][0];

    if (*n_p <= 0 || *n_steps_p < 0 || *delta_t_p
         <= 0) Usage(argv[0]);
```

Osnovna forma algoritma

```
    if (*g_i_p != 'g' && *g_i_p != 'i')
         Usage(argv[0]);

#   ifdef DEBUG
    printf("n = %d\n", *n_p);
    printf("n_steps = %d\n", *n_steps_p);
    printf("delta_t = %e\n", *delta_t_p);
    printf("output_freq = %d\n", *output_freq_p);
```

Osnovna forma algoritma

```
    printf("g_i = %c\n", *g_i_p);
#   endif
}   /* Get_args */


/*--------------------------------------------------
          ---------------------
 * Function:  Get_init_cond
```

Osnovna forma algoritma

```
 * Purpose:   Read in initial conditions:  mass,
         position and velocity
 *            for each particle
 * In args:
 *    n:      number of particles
 * Out args:
 *    curr:   array of n structs, each struct
         stores the mass (scalar),
 *            position (vector), and velocity
         (vector) of a particle
```

Osnovna forma algoritma

```
 */
void Get_init_cond(struct particle_s curr[], int
        n) {
   int part;

   printf("For each particle, enter (in
        order):\n");
   printf("   its mass, its x-coord, its y-coord,
        ");
   printf("its x-velocity, its y-velocity\n");
```

Osnovna forma algoritma

```
   for (part = 0; part < n; part++) {
      scanf("%lf", &curr[part].m);
      scanf("%lf", &curr[part].s[X]);
      scanf("%lf", &curr[part].s[Y]);
      scanf("%lf", &curr[part].v[X]);
      scanf("%lf", &curr[part].v[Y]);
   }
```

Osnovna forma algoritma

```
}  /* Get_init_cond */

/*--------------------------------------------------
        ---------------------
 * Function:  Gen_init_cond
 * Purpose:   Generate initial conditions:  mass,
       position and velocity
 *            for each particle
 * In args:
```

Osnovna forma algoritma

```
*    n:      number of particles
* Out args:
*    curr:   array of n structs, each struct
       stores the mass (scalar),
*            position (vector), and velocity
       (vector) of a particle
*
* Note:      The initial conditions place all
       particles at
*            equal intervals on the nonnegative
       x-axis with
```

Osnovna forma algoritma

```
 *            identical masses, and identical
       initial speeds
 *            parallel to the y-axis.  However,
       some of the
 *            velocities are in the positive y-
       direction and
 *            some are negative.
 */
void Gen_init_cond(struct particle_s curr[], int
       n) {
   int part;
```

Osnovna forma algoritma

```
   double mass = 5.0e24;
   double gap = 1.0e5;
   double speed = 3.0e4;

   srandom(1);
   for (part = 0; part < n; part++) {
      curr[part].m = mass;
```

Osnovna forma algoritma

```
      curr[part].s[X] = part*gap;
      curr[part].s[Y] = 0.0;
      curr[part].v[X] = 0.0;
//    if (random()/((double) RAND_MAX) >= 0.5)
      if (part % 2 == 0)
         curr[part].v[Y] = speed;
      else
```

Osnovna forma algoritma

```
         curr[part].v[Y] = -speed;
   }
}  /* Gen_init_cond */


/*-------------------------------------------------
      ---------------------
 * Function:  Output_state
```

Osnovna forma algoritma

```
 * Purpose:   Print the current state of the
       system
 * In args:
 *    curr:   array with n elements, curr[i]
       stores the state (mass,
 *            position and velocity) of the ith
       particle
 *    n:      number of particles
 */
void Output_state(double time, struct particle_s
       curr[], int n) {
```

Osnovna forma algoritma

```
   int part;
   printf("%.2f\n", time);
   for (part = 0; part < n; part++) {
//    printf("%.3f ", curr[part].m);
      printf("%3d %10.3e ", part,
         curr[part].s[X]);
      printf("  %10.3e ", curr[part].s[Y]);
      printf("  %10.3e ", curr[part].v[X]);
```

Osnovna forma algoritma

```
      printf("  %10.3e\n", curr[part].v[Y]);
   }
   printf("\n");
}  /* Output_state */


/*---------------------------------------------
       ---------------------
```

Osnovna forma algoritma

```
 * Function:  Compute_force
 * Purpose:   Compute the total force on particle
       part.  Exploit
 *            the symmetry (force on particle i
       due to particle k)
 *            = -(force on particle k due to
       particle i) to also
 *            calculate partial forces on other
       particles.
 * In args:
 *    part:   the particle on which we're
       computing the total force
```

Osnovna forma algoritma

```
*    curr:   current state of the system:
       curr[i] stores the mass,
*            position and velocity of the ith
       particle
*    n:      number of particles
* Out arg:
*    forces: force[i] stores the total force on
       the ith particle
*
* Note: This function uses the force due to
       gravitation.  So
```

Osnovna forma algoritma

```
* the force on particle i due to particle k is
       given by
*
*    m_i m_k (s_k - s_i)/|s_k - s_i|^2
*
* Here, m_j is the mass of particle j and s_k is
       its position vector
* (at time t).
*/
```

Osnovna forma algoritma

```
void Compute_force(int part, vect_t forces[],
        struct particle_s curr[],
      int n) {
   int k;
   double mg;
   vect_t f_part_k;
   double len, len_3, fact;
```

Osnovna forma algoritma

```
#  ifdef DEBUG
   printf("Current total force on particle %d =
        (%.3e, %.3e)\n",
         part, forces[part][X], forces[part][Y]);
#  endif
   forces[part][X] = forces[part][Y] = 0.0;
   for (k = 0; k < n; k++) {
      if (k != part) {
```

Osnovna forma algoritma

```
      /* Compute force on part due to k */
         f_part_k[X] = curr[part].s[X] -
        curr[k].s[X];
         f_part_k[Y] = curr[part].s[Y] -
        curr[k].s[Y];
         len = sqrt(f_part_k[X]*f_part_k[X] +
        f_part_k[Y]*f_part_k[Y]);
         len_3 = len*len*len;
         mg = -G*curr[part].m*curr[k].m;
         fact = mg/len_3;
```

Osnovna forma algoritma

```
         f_part_k[X] *= fact;
         f_part_k[Y] *= fact;
#        ifdef DEBUG
         printf("Force on particle %d due to
        particle %d = (%.3e, %.3e)\n",
               part, k, f_part_k[X], f_part_k[Y]);
#        endif
```

Osnovna forma algoritma

```
         /* Add force in to total forces */
         forces[part][X] += f_part_k[X];
         forces[part][Y] += f_part_k[Y];
      }
   }
}  /* Compute_force */
```

Osnovna forma algoritma

```
/*-----------------------------------------------
        ---------------------
 * Function:  Update_part
 * Purpose:   Update the velocity and position for
       particle part
 * In args:
 *    part:    the particle we're updating
 *    forces:  forces[i] stores the total force on
       the ith particle
```

Osnovna forma algoritma

```
 *    n:       number of particles
 *
 * In/out arg:
 *    curr:    curr[i] stores the mass, position
 *             and velocity of the
 *             ith particle
 *
 * Note:  This version uses Euler's method to
 *        update both the velocity
```

Osnovna forma algoritma

```
 *    and the position.
 */
void Update_part(int part, vect_t forces[], struct
        particle_s curr[],
      int n, double delta_t) {
   double fact = delta_t/curr[part].m;

#  ifdef DEBUG
```

Osnovna forma algoritma

```
   printf("Before update of %d:\n", part);
   printf("   Position  = (%.3e, %.3e)\n",
         curr[part].s[X], curr[part].s[Y]);
   printf("   Velocity  = (%.3e, %.3e)\n",
         curr[part].v[X], curr[part].v[Y]);
   printf("   Net force = (%.3e, %.3e)\n",
         forces[part][X], forces[part][Y]);
#  endif
   curr[part].s[X] += delta_t * curr[part].v[X];
   curr[part].s[Y] += delta_t * curr[part].v[Y];
```

Osnovna forma algoritma

```
   curr[part].v[X] += fact * forces[part][X];
   curr[part].v[Y] += fact * forces[part][Y];
#  ifdef DEBUG
   printf("Position of %d = (%.3e, %.3e), Velocity
         = (%.3e,%.3e)\n",
          part, curr[part].s[X], curr[part].s[Y],
                curr[part].v[X], curr[part].v[Y]);
#  endif
```

Osnovna forma algoritma

```
// curr[part].s[X] += delta_t * curr[part].v[X];
// curr[part].s[Y] += delta_t * curr[part].v[Y];
}  /* Update_part */


/*--------------------------------------------------
         ---------------------
 * Function:  Compute_energy
```

Osnovna forma algoritma

```
* Purpose:   Compute the kinetic and potential
      energy in the system
* In args:
*    curr:   current state of the system, curr[i]
      stores the mass,
*            position and velocity of the ith
      particle
*    n:      number of particles
* Out args:
*    kin_en_p: pointer to kinetic energy of
      system
```

Osnovna forma algoritma

```
 *    pot_en_p: pointer to potential energy of
       system
 */
void Compute_energy(struct particle_s curr[], int
        n, double* kin_en_p,
      double* pot_en_p) {
   int i, j;
   vect_t diff;
   double pe = 0.0, ke = 0.0;
```

Osnovna forma algoritma

```
   double dist, speed_sqr;

   for (i = 0; i < n; i++) {
      speed_sqr = curr[i].v[X]*curr[i].v[X] +
        curr[i].v[Y]*curr[i].v[Y];
      ke += curr[i].m*speed_sqr;
   }
   ke *= 0.5;
```

Osnovna forma algoritma

```
   for (i = 0; i < n-1; i++) {
      for (j = i+1; j < n; j++) {
         diff[X] = curr[i].s[X] - curr[j].s[X];
         diff[Y] = curr[i].s[Y] - curr[j].s[Y];
         dist = sqrt(diff[X]*diff[X] +
            diff[Y]*diff[Y]);
         pe += -G*curr[i].m*curr[j].m/dist;
```

Osnovna forma algoritma

```
      }
   }

   *kin_en_p = ke;
   *pot_en_p = pe;
}  /* Compute_energy */
```

Redukovana forma algoritma

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "timer.h"

#define DIM 2  /* Two-dimensional system */
```

Redukovana forma algoritma

```
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */

const double G = 6.673e-11;  /* Gravitational
        constant. */
                             /* Units are
        m^3/(kg*s^2)  */
// const double G = 0.1;  /* Gravitational
        constant. */
                            /* Units are
        m^3/(kg*s^2)  */
```

Redukovana forma algoritma

```
typedef double vect_t[DIM];  /* Vector type for
        position, etc. */

struct particle_s {
   double m;  /* Mass     */
   vect_t s;  /* Position */
   vect_t v;  /* Velocity */
```

Redukovana forma algoritma

```
};

void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p);
void Get_init_cond(struct particle_s curr[], int
        n);
void Gen_init_cond(struct particle_s curr[], int
        n);
```

Redukovana forma algoritma

```
void Output_state(double time, struct particle_s
        curr[], int n);
void Compute_force(int part, vect_t forces[],
        struct particle_s curr[],
      int n);
void Update_part(int part, vect_t forces[], struct
        particle_s curr[],
      int n, double delta_t);
void Compute_energy(struct particle_s curr[], int
        n, double* kin_en_p,
      double* pot_en_p);
```

Redukovana forma algoritma

```
/*--------------------------------------------------
        --------------------*/
int main(int argc, char* argv[]) {
   int n;                    /* Number of
        particles        */
   int n_steps;              /* Number of
        timesteps        */
   int step;                 /* Current step
        */
   int part;                 /* Current particle
        */
```

Redukovana forma algoritma

```
    int output_freq;            /* Frequency of
         output        */
    double delta_t;             /* Size of timestep
         */
    double t;                   /* Current Time
         */
    struct particle_s* curr;    /* Current state of
         system    */
    vect_t* forces;             /* Forces on each
         particle    */
    char g_i;                   /*_G_en or _i_nput
         init conds */
#   ifdef COMPUTE_ENERGY
```

Redukovana forma algoritma

```
    double kinetic_energy, potential_energy;
#   endif
    double start, finish;       /* For timings
         */

    Get_args(argc, argv, &n, &n_steps, &delta_t,
         &output_freq, &g_i);
    curr = malloc(n*sizeof(struct particle_s));
    forces = malloc(n*sizeof(vect_t));
```

Redukovana forma algoritma

```
    if (g_i == 'i')
       Get_init_cond(curr, n);
    else
       Gen_init_cond(curr, n);

    GET_TIME(start);
#   ifdef COMPUTE_ENERGY
```

Redukovana forma algoritma

```
    Compute_energy(curr, n, &kinetic_energy,
         &potential_energy);
    printf("   PE = %e, KE = %e, Total Energy =
         %e\n",
          potential_energy, kinetic_energy,
          kinetic_energy+potential_energy);
#   endif
#   ifndef NO_OUTPUT
    Output_state(0, curr, n);
#   endif
```

Redukovana forma algoritma

```
    for (step = 1; step <= n_steps; step++) {
       t = step*delta_t;
       /* Particle n-1 will have all forces
          computed after call to
        * Compute_force(n-2, . . .) */
       memset(forces, 0, n*sizeof(vect_t));
       for (part = 0; part < n-1; part++)
          Compute_force(part, forces, curr, n);
```

Redukovana forma algoritma

```
       for (part = 0; part < n; part++)
          Update_part(part, forces, curr, n,
          delta_t);
#      ifdef COMPUTE_ENERGY
       Compute_energy(curr, n, &kinetic_energy,
          &potential_energy);
       printf("   PE = %e, KE = %e, Total Energy =
          %e\n",
              potential_energy, kinetic_energy,
          kinetic_energy+potential_energy);
#      endif
```

Redukovana forma algoritma

```
#      ifndef NO_OUTPUT
       if (step % output_freq == 0)
          Output_state(t, curr, n);
#      endif
    }

    GET_TIME(finish);
```

Redukovana forma algoritma

```
    printf("Elapsed time = %e seconds\n", finish-
          start);

    free(curr);
    free(forces);
    return 0;
}   /* main */
```

Redukovana forma algoritma

```
/*---------------------------------------------
        ---------------------
 * Function: Usage
 * Purpose:  Print instructions for command-line
        and exit
 * In arg:
 *    prog_name:  the name of the program as typed
        on the command-line
 */
```

Redukovana forma algoritma

```
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of
        particles> <number of timesteps>\n",
         prog_name);
   fprintf(stderr, "   <size of timestep> <output
        frequency>\n");
   fprintf(stderr, "   <g|i>\n");
   fprintf(stderr, "   'g': program should
        generate init conds\n");
   fprintf(stderr, "   'i': program should get
        init conds from stdin\n");
```

Redukovana forma algoritma

```
   exit(0);
}  /* Usage */


/*---------------------------------------------
        ---------------------
 * Function:  Get_args
```

Redukovana forma algoritma

```
 * Purpose:   Get command line args
 * In args:
 *    argc:            number of command line args
 *    argv:            command line args
 * Out args:
 *    n_p:             pointer to n, the number of
        particles
 *    n_steps_p:       pointer to n_steps, the
        number of timesteps
```

Redukovana forma algoritma

```
 *    delta_t_p:       pointer to delta_t, the
        size of each timestep
 *    output_freq_p:   pointer to output_freq,
        which is the number of
 *                     timesteps between steps
        whose output is printed
 *    g_i_p:           pointer to char which is
        'g' if the init conds
 *                     should be generated by the
        program and 'i' if
 *                     they should be read from
        stdin
 */
```

Redukovana forma algoritma

```
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p) {
   if (argc != 6) Usage(argv[0]);
   *n_p = strtol(argv[1], NULL, 10);
   *n_steps_p = strtol(argv[2], NULL, 10);
   *delta_t_p = strtod(argv[3], NULL);
   *output_freq_p = strtol(argv[4], NULL, 10);
```

Redukovana forma algoritma

```
   *g_i_p = argv[5][0];

   if (*n_p <= 0 || *n_steps_p < 0 || *delta_t_p
        <= 0) Usage(argv[0]);
   if (*g_i_p != 'g' && *g_i_p != 'i')
        Usage(argv[0]);

#  ifdef DEBUG
   printf("n = %d\n", *n_p);
```

Redukovana forma algoritma

```
   printf("n_steps = %d\n", *n_steps_p);
   printf("delta_t = %e\n", *delta_t_p);
   printf("output_freq = %d\n", *output_freq_p);
   printf("g_i = %c\n", *g_i_p);
#  endif
}  /* Get_args */
```

Redukovana forma algoritma

```
/*-----------------------------------------------
        ---------------------
 * Function:  Get_init_cond
 * Purpose:   Read in initial conditions:  mass,
       position and velocity
 *           for each particle
 * In args:
 *    n:      number of particles
```

Redukovana forma algoritma

```
 * Out args:
 *    curr:   array of n structs, each struct
         stores the mass (scalar),
 *           position (vector), and velocity
         (vector) of a particle
 */
void Get_init_cond(struct particle_s curr[], int
        n) {
   int part;
```

Redukovana forma algoritma

```
   printf("For each particle, enter (in
        order):\n");
   printf("   its mass, its x-coord, its y-coord,
        ");
   printf("its x-velocity, its y-velocity\n");
   for (part = 0; part < n; part++) {
      scanf("%lf", &curr[part].m);
      scanf("%lf", &curr[part].s[X]);
      scanf("%lf", &curr[part].s[Y]);
```

Redukovana forma algoritma

```
      scanf("%lf", &curr[part].v[X]);
      scanf("%lf", &curr[part].v[Y]);
   }
}  /* Get_init_cond */

/*-----------------------------------------------
        ---------------------
 * Function:  Gen_init_cond
```

Redukovana forma algoritma

```
 * Purpose:   Generate initial conditions:  mass,
       position and velocity
 *           for each particle
 * In args:
 *    n:      number of particles
 * Out args:
 *    curr:   array of n structs, each struct
       stores the mass (scalar),
 *           position (vector), and velocity
       (vector) of a particle
```

Redukovana forma algoritma

```
 *
 * Note:      The initial conditions place all
       particles at
 *           equal intervals on the nonnegative
       x-axis with
 *           identical masses, and identical
       initial speeds
 *           parallel to the y-axis.  However,
       some of the
 *           velocities are in the positive y-
       direction and
 *           some are negative.
```

Redukovana forma algoritma

```
 */
void Gen_init_cond(struct particle_s curr[], int
       n) {
   int part;
   double mass = 5.0e24;
   double gap = 1.0e5;
   double speed = 3.0e4;
```

Redukovana forma algoritma

```
   srandom(1);
   for (part = 0; part < n; part++) {
      curr[part].m = mass;
      curr[part].s[X] = part*gap;
      curr[part].s[Y] = 0.0;
      curr[part].v[X] = 0.0;
//    if (random()/((double) RAND_MAX) >= 0.5)
```

Redukovana forma algoritma

```
      if (part % 2 == 0)
         curr[part].v[Y] = speed;
      else
         curr[part].v[Y] = -speed;
   }
}  /* Gen_init_cond */
```

Redukovana forma algoritma

```

/*--------------------------------------------------
         ---------------------
 * Function:  Output_state
 * Purpose:   Print the current state of the
       system
 * In args:
 *    curr:   array with n elements, curr[i]
       stores the state (mass,
 *           position and velocity) of the ith
       particle
```

Redukovana forma algoritma

```
 *    n:      number of particles
 */
void Output_state(double time, struct particle_s
        curr[], int n) {
   int part;
   printf("%.2f\n", time);
   for (part = 0; part < n; part++) {
//    printf("%.3f ", curr[part].m);
```

Redukovana forma algoritma

```
      printf("%3d %10.3e ", part,
         curr[part].s[X]);
      printf("  %10.3e ", curr[part].s[Y]);
      printf("  %10.3e ", curr[part].v[X]);
      printf("  %10.3e\n", curr[part].v[Y]);
   }
   printf("\n");
}  /* Output_state */
```

Redukovana forma algoritma

```
/*--------------------------------------------------
        ---------------------
 * Function:  Compute_force
 * Purpose:   Compute the total force on particle
       part.  Exploit
 *           the symmetry (force on particle i
       due to particle k)
 *           = -(force on particle k due to
       particle i) to also
```

Redukovana forma algoritma

```
 *           calculate partial forces on other
       particles.
 * In args:
 *    part:   the particle on which we're
       computing the total force
 *    curr:   current state of the system:
       curr[i] stores the mass,
 *           position and velocity of the ith
       particle
 *    n:      number of particles
 * Out arg:
```

Redukovana forma algoritma

```
 *    forces: force[i] stores the total force on
       the ith particle
 *
 * Note: This function uses the force due to
       gravitation.  So
 * the force on particle i due to particle k is
       given by
 *
 *    m_i m_k (s_k - s_i)/|s_k - s_i|^2
 *
```

Redukovana forma algoritma

```
 * Here, m_j is the mass of particle j and s_k is
       its position vector
 * (at time t).
 */
void Compute_force(int part, vect_t forces[],
        struct particle_s curr[],
      int n) {
   int k;
   double mg;
```

Redukovana forma algoritma

```
    vect_t f_part_k;
    double len, len_3, fact;

#   ifdef DEBUG
    printf("Current total force on particle %d =
         (%.3e, %.3e)\n",
          part, forces[part][X], forces[part][Y]);
#   endif
```

Redukovana forma algoritma

```
    for (k = part+1; k < n; k++) {
       /* Compute force on part due to k */
       f_part_k[X] = curr[part].s[X] -
          curr[k].s[X];
       f_part_k[Y] = curr[part].s[Y] -
          curr[k].s[Y];
       len = sqrt(f_part_k[X]*f_part_k[X] +
          f_part_k[Y]*f_part_k[Y]);
       len_3 = len*len*len;
       mg = -G*curr[part].m*curr[k].m;
```

Redukovana forma algoritma

```
       fact = mg/len_3;
       f_part_k[X] *= fact;
       f_part_k[Y] *= fact;
#      ifdef DEBUG
       printf("Force on particle %d due to particle
         %d = (%.3e, %.3e)\n",
            part, k, f_part_k[X], f_part_k[Y]);
#      endif
```

Redukovana forma algoritma

```
       /* Add force in to total forces */
       forces[part][X] += f_part_k[X];
       forces[part][Y] += f_part_k[Y];
       forces[k][X] -= f_part_k[X];
       forces[k][Y] -= f_part_k[Y];
    }
```

Redukovana forma algoritma

```
}  /* Compute_force */


/*--------------------------------------------------
        ---------------------
 * Function:  Update_part
 * Purpose:   Update the velocity and position for
       particle part
 * In args:
```

Redukovana forma algoritma

```
 *    part:    the particle we're updating
 *    forces:  forces[i] stores the total force on
        the ith particle
 *    n:       number of particles
 *
 * In/out arg:
 *    curr:    curr[i] stores the mass, position
        and velocity of the
 *             ith particle
```

Redukovana forma algoritma

```
 *
 * Note:  This version uses Euler's method to
       update both the velocity
 *    and the position.
 */
void Update_part(int part, vect_t forces[], struct
        particle_s curr[],
      int n, double delta_t) {
   double fact = delta_t/curr[part].m;
```

Redukovana forma algoritma

```
#  ifdef DEBUG
   printf("Before update of %d:\n", part);
   printf("   Position  = (%.3e, %.3e)\n",
        curr[part].s[X], curr[part].s[Y]);
   printf("   Velocity  = (%.3e, %.3e)\n",
        curr[part].v[X], curr[part].v[Y]);
   printf("   Net force = (%.3e, %.3e)\n",
        forces[part][X], forces[part][Y]);
#  endif
```

Redukovana forma algoritma

```
    curr[part].s[X] += delta_t * curr[part].v[X];
    curr[part].s[Y] += delta_t * curr[part].v[Y];
    curr[part].v[X] += fact * forces[part][X];
    curr[part].v[Y] += fact * forces[part][Y];
#   ifdef DEBUG
    printf("Position of %d = (%.3e, %.3e), Velocity
          = (%.3e,%.3e)\n",
           part, curr[part].s[X], curr[part].s[Y],
```

Redukovana forma algoritma

```
                curr[part].v[X], curr[part].v[Y]);
#   endif
// curr[part].s[X] += delta_t * curr[part].v[X];
// curr[part].s[Y] += delta_t * curr[part].v[Y];
}   /* Update_part */
```

Redukovana forma algoritma

```
/*--------------------------------------------------
          ---------------------
 * Function:  Compute_energy
 * Purpose:   Compute the kinetic and potential
          energy in the system
 * In args:
 *    curr:   current state of the system, curr[i]
          stores the mass,
 *           position and velocity of the ith
          particle
 *    n:      number of particles
```

Redukovana forma algoritma

```
 * Out args:
 *    kin_en_p: pointer to kinetic energy of
          system
 *    pot_en_p: pointer to potential energy of
          system
 */
void Compute_energy(struct particle_s curr[], int
          n, double* kin_en_p,
      double* pot_en_p) {
   int i, j;
```

Redukovana forma algoritma

```
    vect_t diff;
    double pe = 0.0, ke = 0.0;
    double dist, speed_sqr;

    for (i = 0; i < n; i++) {
       speed_sqr = curr[i].v[X]*curr[i].v[X] +
          curr[i].v[Y]*curr[i].v[Y];
       ke += curr[i].m*speed_sqr;
```

Redukovana forma algoritma

```
    }
    ke *= 0.5;

    for (i = 0; i < n-1; i++) {
       for (j = i+1; j < n; j++) {
          diff[X] = curr[i].s[X] - curr[j].s[X];
          diff[Y] = curr[i].s[Y] - curr[j].s[Y];
```

Redukovana forma algoritma

```
          dist = sqrt(diff[X]*diff[X] +
             diff[Y]*diff[Y]);
          pe += -G*curr[i].m*curr[j].m/dist;
       }
    }

    *kin_en_p = ke;
    *pot_en_p = pe;
}
```

Diskusija paralelizacije u modelu deljene memorije

Problem n tela

Problemi paralelizacije koda koristeći model deljene memorije

- Suočeni smo sa klasičnim problemom: imamo kod koji radi serijski i hoćemo da ga ubrzamo.
- Ovo je najčešće situacija u kojoj se HPC inženjer nađe, domenski ekspert vam da nekakav kod i pozove vas da ga učinite bržim.
- Mnogo bržim.
- Jedino što se razlikuje od stvarnosti jeste što je šansa da će kod biti urađen u nečemo pogodnijem za brzo prototipiziranje, kao što je Matlab ili, vrlo često ovih dana, Python sa proširenjima za naučno računarstvo.
- U svakom slučaju, kako prići ovakvom problemu?

Fosterova metodolgija

- Particija (eng. Partition)
- Komunikacija (eng. Communication)
- Kombinacija (eng. Agglomeration)
- Mapiranje (eng. Mapping)

Particija

- Particija je podela osnovnog zadatka na pod-zadatke na najprimitivnijem nivou.
- Ovde nas ne zanima nikakvo ograničenje: samo lista stvari koje moramo da uradimo, a koje se na nivou apstrakcije na kome programiramo ne mogu dalje dekomponovati.
- Ako počinjemo od serijskog algoritma, ovaj posao se sastoji od brojanja.

Particija

Komunikacija

- Kada imamo primitivne zadatke, sledeći korak jeste da se ustanovi kakva je komunikacija između njih.
- U modelu deljene memorije to se naročito svodi na graf zavisnosti: koji podatak je potreban za računanje kog podataka.
- Ako se setimo onog fibonačijevog niza od ranije, tu je baš zavisnost između podataka bila deo posla koji nas je ograničavao.

Komunikacija

Kombinacija

- Ako su nam primitivni zadaci potpuno nezavisni, onda možemo da ih kombinujemo kako god želimo.
- U praksi, biće ograničenja, i ta ograničenja vode naše kombinovanje primitivnih zadataka u celine pogodne za obradu.

Kombinacija

Mapiranje

- Konačan korak jeste da kombinovane celine podelimo u grupe koje će se izvršavati na procesnim elementima.
- Mapiranje je takođe podložno ograničenjima, specifično, ograničenjima hardvera/arhitekture.

Mapiranje

.

Primena Fosterove metodologije na problem n tela

- Particija
  - Za svaku česticu (opšti broj i) i korak vremena (opšti broj k) sračunati
    - $\vec{F_i}(t_k - 1)$
    - $\vec{v_i}(t_k)$
    - $\vec{r_i}(t_k)$

Primena Fosterove metodologije na problem n tela

- Komunikacija
  - Za $\vec{v_i}(t_k)$ treba
    - $\vec{v_i}(t_{k-1})$
    - $\vec{F_i}(t_{k-1})$
    - $\vec{r_i}(t_{k-1})$
    - $\vec{r_j}(t_{k-1})$
  - Za $\vec{r_i}(t_k)$ treba
    - $\vec{r_i}(t_{k-1})$
    - $\vec{v_i}(t_{k-1})$

Primena Fosterove metodologije na problem n tela

- Kombinacija
  - Većina komunikacije ide po principu čestica na česticu (i to najviše ista čestica)
  - To znači da su nam kompozitni zadaci prirodno zadaci proračuna za individualnu česticu.
- Mapiranje
  - Imamo dve dimenzija mapiranja: broj čestica i broj koraka.
  - Ojlerova metoda (kao što smo videli u analizi komunikacije) je fundamentalno sekvencijalna.
  - Ovo nameće da paralelizam ima samo smisla ako raspoređujemo čestice po procesima.

Praktično mapiranje

- To znači da, u slučaju modela deljene memorije, na svaku sistemsku nit želimo da blokovski rasporedimo $\frac{n}{p}$ čestica gde je p broj sistemskih niti.
- Zašto blokovski raspored? Zato što smanjuje keš omašaje a posao je homogen i može se lako razdeliti na ovaj način.
- Ovo važi samo ako ne koristimo redukciju da simetrijom smanjimo broj koraka.

Mapiranje u slučaju redukovanog algoritma

- Teškoća je ovde u tome što nisu sada iteracije za svaku vrednost 'i' iste. Niske vrednosti i su mnogo 'skuplje' nego velike vrednosti i.
- Ovo znači da će u blokovskoj podeli da oni koji dobiju visoki blok završiti ranije i morati da besposleno čekaju niže blokove.
- Ovaj scenario je, dakle, dobar za cikličnu distribuciju.
- Ali ciklična distribucija pati od visokog broja omašaja u kešu.
- Pa šta baca više vremena? Nije moguće reći *a priori.* Ovo je jedna od (mnogo) situacija u HPC svetu gde je profilisanje koda neophodno.
- Univerzalno rešenje HPC-a: idi i pogledaj.

Paralelizacija neredukovanog algoritma

- Naivni pristup jeste da se uzmu dve unutarnje petlje algoritma (ona koja računa sile i ona koja računa pomeraje i brzine) i da se for-ovi paralelizuju u statički raspored sa $\frac{n}{p}$ po niti.
- Kako to izgleda?

Paralelizacija neredukovanog algoritma

Potencijalni problemi?

- Potencijalan problem jeste situacija gde paralelne niti mogu da pristupaju istoj promenljivoj na način koji otvara šansu za konflikt, tkzv. ‘race condition.’
- Zato, između ostalog, je komunikacija toliko bitna.
- Kako izgleda pseudo-kod prve petlje?

Analiza prve petlje

Analiza prve petlje

- Za svaki `F[i]` može mu pristupiti samo jedna nit zbog prirode for konstrukta.
- Nit `i` će pristupiti promenljivama `m[j]` i `r[j]` ali ove promenljive se samo čitaju ovde, tako da smo bezbedni. Samo pisanje stvara problem.
- Sve ostale promenljive su privremene i stoga, mogu biti privatne, te su bezbedne.

Analiza druge petlje

Analiza druge petlje

- Isključivo se piše u promenljive `r[i]` i v`[i]` koje su ekskluzivne za odgovarajuću nit.
- Sve ostalo se samo čita.
- Nema mogućeg konflikta.

Redukovani algoritam

Problemi

- Prva petlja nije problem. Može biti paralelizovana manje-više koliko god želimo.
- Druga petlja je problem
- Nit `i` piše i u vrednost F`[i]` i u vrednost F`[j]` što znači da je destruktivno preklapanje između niti apsolutno moguće.
- Ako zamislimo trivijalan primer 4 čestice i 2 niti uz blok-particiju onda računamo silu na treću česticu kao: $ = - - $\vec{f_{1,3}} - \vec{f_{2,3}}$
- Ako, dalje, zamislimo da nit 0 računa prva dva elementa, a nit 1 treći, imamo konflikt. Rezultat zavisi od tajminga.

Naivno rešenje

Naivno rešenje

- Ovo radi, nema problema, ali pretvara naš paralelni kod u serijski u kom slučaju, što smo se trudili?

Malo manje naivno rešenje

Malo manje naivno rešenje

- Ovo je mnogo bolje.
- Nemamo više jedan kritični region koji blokira sve, nego samo tretiramo svaku česticu kao kritični resurs.
- Bolje.
- Ali može još bolje.

Lokalno čuvanje

- Umesto da odmah računamo sve sile i sabiramo ih i smeštamo gde treba mi podelimo posao na dva:
  - Računanje svih sila jedne niti koje smestimo u za nit lokalnu promenljivu.
  - Zbrajanje svih sila jedne čestica tako što se zbrajaju vrednosti lokalnih promenljivih.
- Onda možemo da paralelizujemo ove dve faze odvojeno sa tačkom sinhronizacije između.

Faza 1

- Ovo je isti kod kao i ranije, ali sada svaka nit ima svoje podatke o silama.
- To znači da imamo na kraju malo traćenje memorije i gomilu polu-rezultata.
- Nema konflikta zato što jedino u šta se piše je lokalno za nit.

Faza 1

Faza 2

- Pretvaramo parcijalne u konačne rezultate.
- Možemo o ovome da mislimo kao o ručnoj implementaciji operacije redukcije.
- Nema šanse za konflikt jer i-ta nit je “vlasnik” i-te čestice, a samo se u to piše.

Faza 2

Implementacija paralelizacije u modelu deljene memorije

Problem n tela

OMP Osnovna forma

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <omp.h>

#define DIM 2  /* Two-dimensional system */
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */
```

OMP Osnovna forma

```c
const double G = 6.673e-11;  /* Gravitational
        constant. */
                             /* Units are
        m^3/(kg*s^2)  */
// const double G = 0.1;  /* Gravitational
        constant. */
                        /* Units are m^3/(kg*s^2)
        */

typedef double vect_t[DIM];  /* Vector type for
        position, etc. */
```

OMP Osnovna forma

```
struct particle_s {
   double m;  /* Mass     */
   vect_t s;  /* Position */
   vect_t v;  /* Velocity */
};
```

OMP Osnovna forma

```
void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int*
      thread_count_p, int* n_p,
    int* n_steps_p, double* delta_t_p, int*
      output_freq_p, char* g_i_p);
void Get_init_cond(struct particle_s curr[], int
      n);
void Gen_init_cond(struct particle_s curr[], int
      n);
void Output_state(double time, struct particle_s
      curr[], int n);
void Compute_force(int part, vect_t forces[],
      struct particle_s curr[],
```

OMP Osnovna forma

```
      int n);
void Update_part(int part, vect_t forces[], struct
      particle_s curr[],
      int n, double delta_t);

/*--------------------------------------------
      --------------------*/
int main(int argc, char* argv[]) {
   int n;                      /* Number of
      particles        */
```

OMP Osnovna forma

```
   int n_steps;                /* Number of
      timesteps        */
   int step;                   /* Current step
      */
   int part;                   /* Current particle
      */
   int output_freq;            /* Frequency of
      output           */
   double delta_t;             /* Size of timestep
      */
   double t;                   /* Current Time
      */
   struct particle_s* curr;    /* Current state of
      system    */
```

OMP Osnovna forma

```
   vect_t* forces;             /* Forces on each
        particle    */
   int thread_count;           /* Number of
        threads          */
   char g_i;                   /*_G_en or _i_nput
        init conds */
   double start, finish;       /* For timings
        */

   Get_args(argc, argv, &thread_count, &n,
        &n_steps, &delta_t,
         &output_freq, &g_i);
```

OMP Osnovna forma

```
   curr = malloc(n*sizeof(struct particle_s));
   forces = malloc(n*sizeof(vect_t));
   if (g_i == 'i')
      Get_init_cond(curr, n);
   else
      Gen_init_cond(curr, n);
```

OMP Osnovna forma

```
   start = omp_get_wtime();
#  ifndef NO_OUTPUT
   Output_state(0, curr, n);
#  endif
#  pragma omp parallel num_threads(thread_count)
        default(none) \
      shared(curr, forces, thread_count, delta_t,
        n, n_steps, output_freq) \
      private(step, part, t)
```

OMP Osnovna forma

```
   for (step = 1; step <= n_steps; step++) {
      t = step*delta_t;
//    memset(forces, 0, n*sizeof(vect_t));
#     pragma omp for
      for (part = 0; part < n; part++)
         Compute_force(part, forces, curr, n);
#     pragma omp for
```

OMP Osnovna forma

```
      for (part = 0; part < n; part++)
         Update_part(part, forces, curr, n,
        delta_t);
#     ifndef NO_OUTPUT
#     pragma omp single
      if (step % output_freq == 0)
         Output_state(t, curr, n);
#     endif
```

OMP Osnovna forma

```
   }

   finish = omp_get_wtime();
   printf("Elapsed time = %e seconds\n", finish-
        start);

   free(curr);
   free(forces);
```

OMP Osnovna forma

```
   return 0;
}  /* main */


/*--------------------------------------------------
        ---------------------
 * Function: Usage
 * Purpose:  Print instructions for command-line
        and exit
```

OMP Osnovna forma

```
 * In arg:
 *    prog_name:  the name of the program as typed
        on the command-line
 */
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of threads>
        <number of particles>\n",
         prog_name);
   fprintf(stderr, "   <number of timesteps> <size
        of timestep>\n");
```

OMP Osnovna forma

```
   fprintf(stderr, "   <output frequency>
        <g|i>\n");
   fprintf(stderr, "   'g': program should
        generate init conds\n");
   fprintf(stderr, "   'i': program should get
        init conds from stdin\n");

   exit(0);
}  /* Usage */
```

OMP Osnovna forma

```
/*--------------------------------------------------
        ---------------------
 * Function:  Get_args
 * Purpose:   Get command line args
 * In args:
 *    argc:            number of command line args
 *    argv:            command line args
```

OMP Osnovna forma

```
 * Out args:
 *    thread_count_p:  pointer to thread_count,
        the number of threads
 *    n_p:             pointer to n, the number of
        particles
 *    n_steps_p:       pointer to n_steps, the
        number of timesteps
 *    delta_t_p:       pointer to delta_t, the
        size of each timestep
 *    output_freq_p:   pointer to output_freq,
        which is the number of
 *                     timesteps between steps
        whose output is printed
```

OMP Osnovna forma

```
 *    g_i_p:           pointer to char which is
        'g' if the init conds
 *                     should be generated by the
        program and 'i' if
 *                     they should be read from
        stdin
 */
void Get_args(int argc, char* argv[], int*
        thread_count_p, int* n_p,
      int* n_steps_p, double* delta_t_p, int*
        output_freq_p, char* g_i_p) {
   if (argc != 7) Usage(argv[0]);
```

OMP Osnovna forma

```
   *thread_count_p = strtol(argv[1], NULL, 10);
   *n_p = strtol(argv[2], NULL, 10);
   *n_steps_p = strtol(argv[3], NULL, 10);
   *delta_t_p = strtod(argv[4], NULL);
   *output_freq_p = strtol(argv[5], NULL, 10);
   *g_i_p = argv[6][0];
```

OMP Osnovna forma

```
   if (*thread_count_p < 0 || *n_p <= 0 ||
        *n_steps_p < 0 || *delta_t_p <= 0)
      Usage(argv[0]);
   if (*g_i_p != 'g' && *g_i_p != 'i')
        Usage(argv[0]);

#  ifdef DEBUG
   printf("thread_count = %d\n", *thread_count_p);
   printf("n = %d\n", *n_p);
```

OMP Osnovna forma

```
   printf("n_steps = %d\n", *n_steps_p);
   printf("delta_t = %e\n", *delta_t_p);
   printf("output_freq = %d\n", *output_freq_p);
   printf("g_i = %c\n", *g_i_p);
#  endif
}  /* Get_args */
```

OMP Osnovna forma

```
/*---------------------------------------------------
         ---------------------
 * Function:  Get_init_cond
 * Purpose:   Read in initial conditions:  mass,
        position and velocity
 *            for each particle
 * In args:
 *    n:      number of particles
```

OMP Osnovna forma

```
 * Out args:
 *    curr:   array of n structs, each struct
          stores the mass (scalar),
 *           position (vector), and velocity
         (vector) of a particle
 */
void Get_init_cond(struct particle_s curr[], int
         n) {
   int part;
```

OMP Osnovna forma

```
   printf("For each particle, enter (in
        order):\n");
   printf("   its mass, its x-coord, its y-coord,
        ");
   printf("its x-velocity, its y-velocity\n");
   for (part = 0; part < n; part++) {
      scanf("%lf", &curr[part].m);
      scanf("%lf", &curr[part].s[X]);
      scanf("%lf", &curr[part].s[Y]);
```

OMP Osnovna forma

```
      scanf("%lf", &curr[part].v[X]);
      scanf("%lf", &curr[part].v[Y]);
   }
}  /* Get_init_cond */

/*--------------------------------------------------
        ---------------------
 * Function:  Gen_init_cond
```

OMP Osnovna forma

```
* Purpose:   Generate initial conditions:  mass,
      position and velocity
*           for each particle
* In args:
*    n:      number of particles
* Out args:
*    curr:   array of n structs, each struct
      stores the mass (scalar),
*           position (vector), and velocity
      (vector) of a particle
```

OMP Osnovna forma

```
 *
 * Note:      The initial conditions place all
       particles at
 *            equal intervals on the nonnegative
       x-axis with
 *            identical masses, and identical
       initial speeds
 *            parallel to the y-axis.  However,
       some of the
 *            velocities are in the positive y-
       direction and
 *            some are negative.
```

OMP Osnovna forma

```
 */
void Gen_init_cond(struct particle_s curr[], int
        n) {
   int part;
   double mass = 5.0e24;
   double gap = 1.0e5;
   double speed = 3.0e4;
```

OMP Osnovna forma

```
   srandom(1);
   for (part = 0; part < n; part++) {
      curr[part].m = mass;
      curr[part].s[X] = part*gap;
      curr[part].s[Y] = 0.0;
      curr[part].v[X] = 0.0;
//    if (random()/((double) RAND_MAX) >= 0.5)
```

OMP Osnovna forma

```
      if (part % 2 == 0)
         curr[part].v[Y] = speed;
      else
         curr[part].v[Y] = -speed;
   }
}  /* Gen_init_cond */
```

OMP Osnovna forma

```
/*--------------------------------------------------
        ---------------------
 * Function:  Output_state
 * Purpose:   Print the current state of the
       system
 * In args:
 *    curr:   array with n elements, curr[i]
       stores the state (mass,
 *           position and velocity) of the ith
       particle
```

OMP Osnovna forma

```
 *    n:      number of particles
 */
void Output_state(double time, struct particle_s
        curr[], int n) {
   int part;
   printf("%.2f\n", time);
   for (part = 0; part < n; part++) {
//    printf("%.3f ", curr[part].m);
```

OMP Osnovna forma

```
      printf("%3d %10.3e ", part,
        curr[part].s[X]);
      printf("  %10.3e ", curr[part].s[Y]);
      printf("  %10.3e ", curr[part].v[X]);
      printf("  %10.3e\n", curr[part].v[Y]);
   }
   printf("\n");
}  /* Output_state */
```

OMP Osnovna forma

```
/*--------------------------------------------------
        ---------------------
 * Function:  Compute_force
 * Purpose:   Compute the total force on particle
       part.
 * In args:
 *    part:   the particle on which we're
       computing the total force
```

OMP Osnovna forma

```
 *    curr:   current state of the system:
 *              curr[i] stores the mass,
 *              position and velocity of the ith
 *              particle
 *    n:      number of particles
 * Out arg:
 *    forces: force[i] stores the total force on
 *              the ith particle
 *
 * Note: This function uses the force due to
 *       gravitation.  So
```

OMP Osnovna forma

```
 * the force on particle i due to particle k is
 *       given by
 *
 *    m_i m_k (s_k - s_i)/|s_k - s_i|^2
 *
 * Here, m_j is the mass of particle j and s_k is
 *       its position vector
 * (at time t).
 */
```

OMP Osnovna forma

```
void Compute_force(int part, vect_t forces[],
        struct particle_s curr[],
      int n) {
   int k;
   double mg;
   vect_t f_part_k;
   double len, len_3, fact;
```

OMP Osnovna forma

```
#  ifdef DEBUG
   printf("Current total force on particle %d =
        (%.3e, %.3e)\n",
         part, forces[part][X], forces[part][Y]);
#  endif
   forces[part][X] = forces[part][Y] = 0.0;
   for (k = 0; k < n; k++) {
      if (k != part) {
```

OMP Osnovna forma

```
      /* Compute force on part due to k */
         f_part_k[X] = curr[part].s[X] -
        curr[k].s[X];
         f_part_k[Y] = curr[part].s[Y] -
        curr[k].s[Y];
         len = sqrt(f_part_k[X]*f_part_k[X] +
        f_part_k[Y]*f_part_k[Y]);
         len_3 = len*len*len;
         mg = -G*curr[part].m*curr[k].m;
         fact = mg/len_3;
```

OMP Osnovna forma

```
         f_part_k[X] *= fact;
         f_part_k[Y] *= fact;
#        ifdef DEBUG
         printf("Force on particle %d due to
        particle %d = (%.3e, %.3e)\n",
               part, k, f_part_k[X], f_part_k[Y]);
#        endif
```

OMP Osnovna forma

```
         /* Add force in to total forces */
         forces[part][X] += f_part_k[X];
         forces[part][Y] += f_part_k[Y];
      }
   }
}  /* Compute_force */
```

OMP Osnovna forma

```
/*--------------------------------------------------
         ---------------------
 * Function:  Update_part
 * Purpose:   Update the velocity and position for
       particle part
 * In args:
 *    part:    the particle we're updating
 *    forces:  forces[i] stores the total force on
       the ith particle
```

OMP Osnovna forma

```
 *    n:       number of particles
 *
 * In/out arg:
 *    curr:    curr[i] stores the mass, position
 *             and velocity of the
 *             ith particle
 *
 * Note:  This version uses Euler's method to
 *        update both the velocity
```

OMP Osnovna forma

```
 *    and the position.
 */
void Update_part(int part, vect_t forces[], struct
        particle_s curr[],
      int n, double delta_t) {
   double fact = delta_t/curr[part].m;

#  ifdef DEBUG
```

OMP Osnovna forma

```
   printf("Before update of %d:\n", part);
   printf("   Position  = (%.3e, %.3e)\n",
        curr[part].s[X], curr[part].s[Y]);
   printf("   Velocity  = (%.3e, %.3e)\n",
        curr[part].v[X], curr[part].v[Y]);
   printf("   Net force = (%.3e, %.3e)\n",
        forces[part][X], forces[part][Y]);
#  endif
   curr[part].s[X] += delta_t * curr[part].v[X];
   curr[part].s[Y] += delta_t * curr[part].v[Y];
```

OMP Osnovna forma

```
   curr[part].v[X] += fact * forces[part][X];
   curr[part].v[Y] += fact * forces[part][Y];
#  ifdef DEBUG
   printf("Position of %d = (%.3e, %.3e), Velocity
        = (%.3e,%.3e)\n",
         part, curr[part].s[X], curr[part].s[Y],
               curr[part].v[X], curr[part].v[Y]);
#  endif
```

OMP Osnovna forma

```
// curr[part].s[X] += delta_t * curr[part].v[X];
// curr[part].s[Y] += delta_t * curr[part].v[Y];
}  /* Update_part */


/*--------------------------------------------------
        ---------------------
 * Function:  Compute_energy
```

OMP Osnovna forma

```
* Purpose:   Compute the kinetic and potential
      energy in the system
* In args:
*    curr:   current state of the system, curr[i]
      stores the mass,
*            position and velocity of the ith
      particle
*    n:      number of particles
* Out args:
*    kin_en_p: pointer to kinetic energy of
      system
```

OMP Osnovna forma

```
 *    pot_en_p: pointer to potential energy of
       system
 */
void Compute_energy(struct particle_s curr[], int
        n, double* kin_en_p,
      double* pot_en_p) {
   int i, j;
   vect_t diff;
   double pe = 0.0, ke = 0.0;
```

OMP Osnovna forma

```
   double dist, speed_sqr;

   for (i = 0; i < n; i++) {
      speed_sqr = curr[i].v[X]*curr[i].v[X] +
        curr[i].v[Y]*curr[i].v[Y];
      ke += curr[i].m*speed_sqr;
   }
   ke *= 0.5;
```

OMP Osnovna forma

```
   for (i = 0; i < n-1; i++) {
      for (j = i+1; j < n; j++) {
         diff[X] = curr[i].s[X] - curr[j].s[X];
         diff[Y] = curr[i].s[Y] - curr[j].s[Y];
         dist = sqrt(diff[X]*diff[X] +
            diff[Y]*diff[Y]);
         pe += -G*curr[i].m*curr[j].m/dist;
```

OMP Osnovna forma

```
      }
   }

   *kin_en_p = ke;
   *pot_en_p = pe;
}  /* Compute_energy */
```

OMP Redukovana forma

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <omp.h>
```

OMP Redukovana forma

```
#define DIM 2  /* Two-dimensional system */
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */

const double G = 6.673e-11;  /* Gravitational
        constant. */
                             /* Units are
        m^3/(kg*s^2)  */
```

OMP Redukovana forma

```
typedef double vect_t[DIM];  /* Vector type for
        position, etc. */

struct particle_s {
   double m;  /* Mass     */
   vect_t s;  /* Position */
   vect_t v;  /* Velocity */
};
```

OMP Redukovana forma

```
void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int*
        thread_count_p, int* n_p,
      int* n_steps_p, double* delta_t_p, int*
        output_freq_p, char* g_i_p);
void Get_init_cond(struct particle_s curr[], int
        n);
void Gen_init_cond(struct particle_s curr[], int
        n);
void Output_state(double time, struct particle_s
        curr[], int n);
```

OMP Redukovana forma

```
void Compute_force(int part, vect_t forces[],
        struct particle_s curr[],
      int n);
void Update_part(int part, vect_t forces[], struct
        particle_s curr[],
      int n, double delta_t);

/*--------------------------------------------------
        --------------------*/
int main(int argc, char* argv[]) {
```

OMP Redukovana forma

```
   int n;                    /* Number of
        particles             */
   int n_steps;              /* Number of
        timesteps             */
   int step;                 /* Current step
        */
   int part;                 /* Current particle
        */
   int output_freq;          /* Frequency of
        output             */
   double delta_t;           /* Size of timestep
        */
   double t;                 /* Current Time
        */
```

OMP Redukovana forma

```
   struct particle_s* curr;     /* Current state of
        system           */
   vect_t* forces;              /* Forces on each
        particle         */
   int thread_count;            /* Number of
        threads                 */
   char g_i;                    /* _G_enerate or
        _i_nput init conds */
   double start, finish;        /* For timing
        */
   vect_t* loc_forces;          /* Forces computed
        by each thread   */
```

OMP Redukovana forma

```
   Get_args(argc, argv, &thread_count, &n,
        &n_steps, &delta_t,
         &output_freq, &g_i);
   curr = malloc(n*sizeof(struct particle_s));
   forces = malloc(n*sizeof(vect_t));
   loc_forces =
        malloc(thread_count*n*sizeof(vect_t));
   if (g_i == 'i')
      Get_init_cond(curr, n);
```

OMP Redukovana forma

```
   else
      Gen_init_cond(curr, n);

   start = omp_get_wtime();
#  ifndef NO_OUTPUT
   Output_state(0, curr, n);
#  endif
```

OMP Redukovana forma

```
#  pragma omp parallel num_threads(thread_count)
        default(none) \
      shared(curr,forces,thread_count,delta_t,n,n_st
           output_freq,loc_forces) \
      private(step, part, t)
   {
      int my_rank = omp_get_thread_num();
      int thread;
```

OMP Redukovana forma

```
      for (step = 1; step <= n_steps; step++) {
         t = step*delta_t;
//       memset(loc_forces + my_rank*n, 0,
        n*sizeof(vect_t));
#        pragma omp for
         for (part = 0; part < thread_count*n;
        part++)
            loc_forces[part][X] = loc_forces[part]
        [Y] = 0.0;
```

OMP Redukovana forma

```
#        ifdef DEBUG
#        pragma omp single
         {
            printf("Step %d, after memset
        loc_forces = \n", step);
            for (part = 0; part < thread_count*n;
        part++)
               printf("%d %e %e\n", part,
        loc_forces[part][X],
                  loc_forces[part][Y]);
```

OMP Redukovana forma

```
            printf("\n");
         }
#        endif
         /* Particle n-1 will have all forces
        computed after call to
          * Compute_force(n-2, . . .) */
#        pragma omp for schedule(static,1)
         for (part = 0; part < n-1; part++)
```

OMP Redukovana forma

```
            Compute_force(part, loc_forces +
        my_rank*n, curr, n);
#        pragma omp for
         for (part = 0; part < n; part++) {
            forces[part][X] = forces[part][Y] =
        0.0;
            for (thread = 0; thread <
        thread_count; thread++) {
               forces[part][X] +=
        loc_forces[thread*n + part][X];
               forces[part][Y] +=
        loc_forces[thread*n + part][Y];
```

OMP Redukovana forma

```
            }
         }
#        pragma omp for
         for (part = 0; part < n; part++)
            Update_part(part, forces, curr, n,
        delta_t);
#        ifndef NO_OUTPUT
         if (step % output_freq == 0) {
```

OMP Redukovana forma

```
#           pragma omp single
            Output_state(t, curr, n);
         }
#        endif
      }  /* for step */
   }  /* pragma omp parallel */
   finish = omp_get_wtime();
```

OMP Redukovana forma

```
   printf("Elapsed time = %e seconds\n", finish-
        start);

   free(curr);
   free(forces);
   free(loc_forces);
   return 0;
}  /* main */
```

OMP Redukovana forma

```
/*--------------------------------------------------
        ---------------------
 * Function: Usage
 * Purpose:  Print instructions for command-line
       and exit
 * In arg:
 *    prog_name:  the name of the program as typed
       on the command-line
 */
```

OMP Redukovana forma

```
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of threads>
        <number of particles>\n",
         prog_name);
   fprintf(stderr, "   <number of timesteps>
        <size of timestep>\n");
   fprintf(stderr, "   <output frequency>
        <g|i>\n");
   fprintf(stderr, "   'g': program should
        generate init conds\n");
   fprintf(stderr, "   'i': program should get
        init conds from stdin\n");
```

OMP Redukovana forma

```
   exit(0);
}  /* Usage */


/*--------------------------------------------------
         ---------------------
 * Function:  Get_args
```

OMP Redukovana forma

```
 * Purpose:   Get command line args
 * In args:
 *    argc:           number of command line args
 *    argv:           command line args
 * Out args:
 *    thread_count_p:  pointer to thread_count
 *    n_p:            pointer to n, the number of
       particles
```

OMP Redukovana forma

```
 *    n_steps_p:       pointer to n_steps, the
       number of timesteps
 *    delta_t_p:       pointer to delta_t, the
       size of each timestep
 *    output_freq_p:   pointer to output_freq,
       which is the number of
 *                     timesteps between steps
       whose output is printed
 *    g_i_p:           pointer to char which is
       'g' if the init conds
 *                     should be generated by the
       program and 'i' if
 *                     they should be read from
       stdin
```

OMP Redukovana forma

```
 */
void Get_args(int argc, char* argv[], int*
        thread_count_p, int* n_p,
      int* n_steps_p, double* delta_t_p, int*
        output_freq_p,
      char* g_i_p) {
   if (argc != 7) Usage(argv[0]);
   *thread_count_p = strtol(argv[1], NULL, 10);
   *n_p = strtol(argv[2], NULL, 10);
```

OMP Redukovana forma

```
   *n_steps_p = strtol(argv[3], NULL, 10);
   *delta_t_p = strtod(argv[4], NULL);
   *output_freq_p = strtol(argv[5], NULL, 10);
   *g_i_p = argv[6][0];

   if (*thread_count_p <= 0 || *n_p <= 0 ||
        *n_steps_p < 0 ||
       *delta_t_p <= 0) Usage(argv[0]);
```

OMP Redukovana forma

```
   if (*g_i_p != 'g' && *g_i_p != 'i')
        Usage(argv[0]);

#  ifdef DEBUG
   printf("thread_count = %d\n", *thread_count_p);
   printf("n = %d\n", *n_p);
   printf("n_steps = %d\n", *n_steps_p);
   printf("delta_t = %e\n", *delta_t_p);
```

OMP Redukovana forma

```
   printf("output_freq = %d\n", *output_freq_p);
   printf("g_i = %c\n", *g_i_p);
#  endif
}  /* Get_args */

/*--------------------------------------------------
      ---------------------
 * Function:  Get_init_cond
```

OMP Redukovana forma

```
* Purpose:   Read in initial conditions:  mass,
      position and velocity
*           for each particle
* In args:
*    n:      number of particles
* Out args:
*    curr:   array of n structs, each struct
      stores the mass (scalar),
*           position (vector), and velocity
      (vector) of a particle
```

OMP Redukovana forma

```
 */
void Get_init_cond(struct particle_s curr[], int
      n) {
   int part;

   printf("For each particle, enter (in
      order):\n");
   printf("   its mass, its x-coord, its y-coord,
      ");
   printf("its x-velocity, its y-velocity\n");
```

OMP Redukovana forma

```
   for (part = 0; part < n; part++) {
      scanf("%lf", &curr[part].m);
      scanf("%lf", &curr[part].s[X]);
      scanf("%lf", &curr[part].s[Y]);
      scanf("%lf", &curr[part].v[X]);
      scanf("%lf", &curr[part].v[Y]);
   }
```

OMP Redukovana forma

```
}  /* Get_init_cond */

/*--------------------------------------------
      ---------------------
 * Function:  Gen_init_cond
 * Purpose:   Generate initial conditions:  mass,
      position and velocity
 *           for each particle
 * In args:
```

OMP Redukovana forma

```
*    n:      number of particles
* Out args:
*    curr:   array of n structs, each struct
       stores the mass (scalar),
*            position (vector), and velocity
       (vector) of a particle
*
* Note:      The initial conditions place all
       particles at
*            equal intervals on the nonnegative
       x-axis with
```

OMP Redukovana forma

```
 *           identical masses, and identical
       initial speeds
 *           parallel to the y-axis.  However,
       some of the
 *           velocities are in the positive y-
       direction and
 *           some are negative.
 */
void Gen_init_cond(struct particle_s curr[], int
       n) {
   int part;
```

OMP Redukovana forma

```
   double mass = 5.0e24;
   double gap = 1.0e5;
   double speed = 3.0e4;

   srandom(1);
   for (part = 0; part < n; part++) {
      curr[part].m = mass;
```

OMP Redukovana forma

```
      curr[part].s[X] = part*gap;
      curr[part].s[Y] = 0.0;
      curr[part].v[X] = 0.0;
//    if (random()/((double) RAND_MAX) >= 0.5)
      if (part % 2 == 0)
         curr[part].v[Y] = speed;
      else
```

OMP Redukovana forma

```
         curr[part].v[Y] = -speed;
      }
   }  /* Gen_init_cond */

   /*---------------------------------------------
           ---------------------
    * Function:  Output_state
    * Purpose:   Print the current state of the
          system
```

OMP Redukovana forma

```
    * In args:
    *    curr:   array with n elements, curr[i]
           stores the state (mass,
    *           position and velocity) of the ith
           particle
    *    n:      number of particles
    */
   void Output_state(double time, struct particle_s
           curr[], int n) {
      int part;
```

OMP Redukovana forma

```
      printf("%.2f\n", time);
      for (part = 0; part < n; part++) {
   //    printf("%.3e ", curr[part].m);
         printf("%3d %10.3e ", part,
            curr[part].s[X]);
         printf("  %10.3e ", curr[part].s[Y]);
         printf("  %10.3e ", curr[part].v[X]);
         printf("  %10.3e\n", curr[part].v[Y]);
```

OMP Redukovana forma

```
      }
      printf("\n");
   }  /* Output_state */


   /*---------------------------------------------
           ---------------------
    * Function:  Compute_force
```

OMP Redukovana forma

```
 * Purpose:   Compute the total force on particle
       part.  Exploit
 *            the symmetry (force on particle i
       due to particle k)
 *           = -(force on particle k due to
       particle i) to also
 *            calculate partial forces on other
       particles.
 * In args:
 *    part:   the particle on which we're
       computing the total force
 *    curr:   current state of the system:
       curr[i] stores the mass,
```

OMP Redukovana forma

```
 *            position and velocity of the ith
       particle
 *    n:      number of particles
 * Out arg:
 *    forces: force[i] stores the total force on
       the ith particle
 *
 * Note: This function uses the force due to
       gravitation.  So
 * the force on particle i due to particle k is
       given by
```

OMP Redukovana forma

```
 *
 *    m_i m_k (s_k - s_i)/|s_k - s_i|^2
 *
 * Here, m_j is the mass of particle j and s_k is
       its position vector
 * (at time t).
 */
void Compute_force(int part, vect_t forces[],
       struct particle_s curr[],
```

OMP Redukovana forma

```
      int n) {
   int k;
   double mg;
   vect_t f_part_k;
   double len, len_3, fact;

#  ifdef DEBUG
```

OMP Redukovana forma

```
      printf("Current total force on particle %d =
        (%.3e, %.3e)\n",
          part, forces[part][X], forces[part][Y]);
#     endif
      for (k = part+1; k < n; k++) {
        /* Compute force on part due to k */
        f_part_k[X] = curr[part].s[X] -
          curr[k].s[X];
        f_part_k[Y] = curr[part].s[Y] -
          curr[k].s[Y];
```

OMP Redukovana forma

```
        len = sqrt(f_part_k[X]*f_part_k[X] +
          f_part_k[Y]*f_part_k[Y]);
        len_3 = len*len*len;
        mg = -G*curr[part].m*curr[k].m;
        fact = mg/len_3;
        f_part_k[X] *= fact;
        f_part_k[Y] *= fact;
#       ifdef DEBUG
```

OMP Redukovana forma

```
        printf("Force on particle %d due to particle
          %d = (%.3e, %.3e)\n",
            part, k, f_part_k[X], f_part_k[Y]);
#       endif

        /* Add force into total forces */
        forces[part][X] += f_part_k[X];
        forces[part][Y] += f_part_k[Y];
```

OMP Redukovana forma

```
        forces[k][X] -= f_part_k[X];
        forces[k][Y] -= f_part_k[Y];
      }
    }  /* Compute_force */


    /*--------------------------------------------------
            ---------------------
```

OMP Redukovana forma

```
 * Function:  Update_part
 * Purpose:   Update the velocity and position for
       particle part
 * In args:
 *    part:    the particle we're updating
 *    forces:  forces[] stores the total force on
       the ith particle
 *    n:       number of particles
 *
```

OMP Redukovana forma

```
 * In/out arg:
 *    curr:    curr[i] stores the mass, position
       and velocity of the
 *             ith particle
 *
 * Note:  This version uses Euler's method to
       update both the velocity
 *    and the position.
 */
```

OMP Redukovana forma

```
void Update_part(int part, vect_t forces[], struct
      particle_s curr[],
      int n, double delta_t) {
   double fact = delta_t/curr[part].m;

#  ifdef DEBUG
   printf("Before update of %d:\n", part);
   printf("   Position  = (%.3e, %.3e)\n",
      curr[part].s[X], curr[part].s[Y]);
```

OMP Redukovana forma

```
   printf("   Velocity  = (%.3e, %.3e)\n",
      curr[part].v[X], curr[part].v[Y]);
   printf("   Net force = (%.3e, %.3e)\n",
      forces[part][X], forces[part][Y]);
#  endif
   curr[part].s[X] += delta_t * curr[part].v[X];
   curr[part].s[Y] += delta_t * curr[part].v[Y];
   curr[part].v[X] += fact * forces[part][X];
   curr[part].v[Y] += fact * forces[part][Y];
```

OMP Redukovana forma

```
#  ifdef DEBUG
   printf("Position of %d = (%.3e, %.3e), Velocity
        = (%.3e,%.3e)\n",
          part, curr[part].s[X], curr[part].s[Y],
                curr[part].v[X], curr[part].v[Y]);
#  endif
}
```

Diskusija paralelizacije u modelu distribuirane memorije

Problem n tela

OpenMPI paralelizacija

- Centralna ideja se ne razlikuje dramatično u odnosu na OpenMP paralelizaciju.
- Ključan korak, kao i ranije, jeste kako individualne niti (koje su sada na različitim mašinama) komuniciraju.
- Fundamentalna podela posla, opet, radi na nivou individualnih čestica.
- Ako govorimo o ne-redukovanom algoritmu, onda komunikacija nastaje u okviru računanja sile između i-te i j-te čestice.
- To znači da u svakom koraku, svaka čestica zahteva poziciju svake druge čestice (manje-više).
- Ovo je dušu dalo za `MPI_Allgather` operaciju.

Tehnički detalji struktura podataka

- Pod MPI umesto struct-a koristimo nizove elementarnih tipova zato što je komunikacija kroz elementarne tipove najbrža.
- Takođe, niz masa držimo unapred kopiranim na sve instance zato što se mase nikad ne menjaju, a uvek su neophodne.
- Najbolji način da razrešimo distribuciju podataka jeste kroz blokove pozicija za koje smo odgovorni (pozicije su ono što moramo da delimo iz koraka u korak) gde svaka nit pamti koji je njen blok kroz pokazivače i fiksnu veličinu bloka.

Opšta struktura ne-redukovanog algoritma

MPI implementacija redukovanog algoritma

- Redukovani algoritam je jako problematičan za implementaciju u MPI okruženju.
- Problem je kako proslediti prave informacije u pravom trenutku.
  - Svaki proces je odgovoran za neke čestice, ali ne računa sve sile za te čestice. Neke sile će računati neko drugi.
  - To znači da u svakom koraku moramo i da emitujemo naše sile drugim procesima i da dobijemo sile od negde drugde samo da bi mogli da sračunamo sledeću generaciju pozicija.
- Ovo je jako kompleksno, ali ako se implementira kako treba je šansa za istinski skalabilan i efikasan proces za računanje proizvoljno velikih n-tela simulacija.

Prstenska komunikaciona šema

- Prstenska komunikaciona šema je stari trik u komunikacionoj topologiji da se pojednostavi i regularizuje komunikacija.
- U praksi, svaki čvor može da komunicira sa svakim drugim čvorom na bilo koji način, bilo kada, sa bilo kojim sadržajem.
- Ova sloboda znači da ništa ne može, *a priori,* da se kaže o toj komunikaciji, ona nema osobine, i o njoj se ne može baš puno rezonovati: suviše je nedefinisano.
- Problem do koga ovo ultimativno dovede jeste da ne možemo da optimizujemo komunikaciju zato što je ne razumemo.
- Rešenje jeste da se koriste obrasci gde ograničenja stvaraju strukturu.

Prstenska komunikaciona šema

- U ovoj šemi zamislimo sve MPI procese (njih q) u prstenu.
- Zatim ih povežemo vodeći računa o ograničenju da p-ti proces samo komunicira sa procesima `p-1` i `p+1` ili, da bi bili nešto tačniji, sa `(p - 1 + q) % q` i `(p+1) % q` (što se samo stara da se očuva prstenasta topologija.
- U okviru komunikacije važi pipeline princip: komunikacija je sinhronizovana i to tako da `p` prima podatke od `p-1` i šalje ih `p+1`
- Posle q faza komunikacije, svi imaju sve podatke.

Paralelizacija redukovanog algoritma kroz MPI

- Neka je broj čestica dodeljenih jednom procesu `l`
- Onda u našoj prstenskoj strukturi u svakoj fazi komunikacije svaki proces:
  - Šalje dalje l sila i l pozicija (svojih)
  - Proračuna silu između svojih čestica i onih čije pozicije primi.
  - Doda te sile svojim česticama i oduzme ih od sila koje dobije.

Primer prstenskog stila komunikacije za n=4, p=2

Primer prstenskog stila komunikacije za n=4, p=2

Tehnički detalji komunikacije

- Dovoljno je često programirati ovako da se podaci istovremeno šalju i primaju da postoji funkcija koja nam štedi memoriju za takav tip komunikacije: `MPI_Sendrecv_replace` čiji su parametri:
  - ulazno izlazni bafer
  - veličina bafera
  - tip podataka
  - odredište
  - tag
  - izvor
  - tag
  - kanal
  - izlazni status

`MPI_Sendrecv_replace`

- `MPI_Sendrecv_replace` istovremeno šalje podatke u baferu i prima istu količinu podataka.
- Kada se završi, podaci koji su bili u baferu su odaslati dalje, a podaci koji su stigli su smešteni u bafer.

Ako želite još informacija

- Vrlo dobar pregled problema N tela se može naći u Grop, Lusk, i Skjellum — Using MPI: Portable Parallel Programming with the Message Passing Interface u sekciji 5.2.
- Više o efikasnoj integraciji diferencijalnih jednačina ima u: William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. Numerical Recipes 3rd Edition: The Art of Scientific Computing (3 ed.). Cambridge University Press, New York, NY, USA. i to u poglavlju 17, stranica 899. Naročito obratite pažnju na sekcije 17.0 i 17.1.

Implementacija paralelizacije u modelu distribuirane memorije

Problem n tela

MPI Osnovna forma

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>
```

MPI Osnovna forma

```
#define DIM 2  /* Two-dimensional system */
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */

typedef double vect_t[DIM];  /* Vector type for
        position, etc. */

/* Global variables.  Except or vel all are
        unchanged after being set */
```

MPI Osnovna forma

```
const double G = 6.673e-11;  /* Gravitational
        constant. */
                             /* Units are
        m^3/(kg*s^2)  */
int my_rank, comm_sz;
MPI_Comm comm;
MPI_Datatype vect_mpi_t;

/* Scratch array used by process 0 for global
        velocity I/O */
```

MPI Osnovna forma

```
vect_t *vel = NULL;

void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p);
void Get_init_cond(double masses[], vect_t pos[],
      vect_t loc_vel[], int n, int loc_n);
```

MPI Osnovna forma

```
void Gen_init_cond(double masses[], vect_t pos[],
      vect_t loc_vel[], int n, int loc_n);
void Output_state(double time, double masses[],
        vect_t pos[],
      vect_t loc_vel[], int n, int loc_n);
void Compute_force(int loc_part, double masses[],
        vect_t loc_forces[],
      vect_t pos[], int n, int loc_n);
void Update_part(int loc_part, double masses[],
        vect_t loc_forces[],
```

MPI Osnovna forma

```
      vect_t loc_pos[], vect_t loc_vel[], int n,
        int loc_n, double delta_t);

/*---------------------------------------------
        --------------------*/
int main(int argc, char* argv[]) {
   int n;                    /* Total number of
        particles  */
   int loc_n;                /* Number of my
        particles     */
   int n_steps;              /* Number of
        timesteps        */
```

MPI Osnovna forma

```
   int step;                 /* Current step
        */
   int loc_part;             /* Current local
        particle     */
   int output_freq;          /* Frequency of
        output         */
   double delta_t;           /* Size of timestep
        */
   double t;                 /* Current Time
        */
   double* masses;           /* All the masses
        */
   vect_t* loc_pos;          /* Positions of my
        particles  */
```

MPI Osnovna forma

```
   vect_t* pos;              /* Positions of all
        particles */
   vect_t* loc_vel;          /* Velocities of my
        particles */
   vect_t* loc_forces;       /* Forces on my
        particles     */

   char g_i;                 /*_G_en or _i_nput
        init conds */
   double start, finish;     /* For timings
        */
```

MPI Osnovna forma

```
   MPI_Init(&argc, &argv);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &comm_sz);
   MPI_Comm_rank(comm, &my_rank);

   Get_args(argc, argv, &n, &n_steps, &delta_t,
        &output_freq, &g_i);
   loc_n = n/comm_sz;  /* n should be evenly
        divisible by comm_sz */
```

MPI Osnovna forma

```
   masses = malloc(n*sizeof(double));
   pos = malloc(n*sizeof(vect_t));
   loc_forces = malloc(loc_n*sizeof(vect_t));
   loc_pos = pos + my_rank*loc_n;
   loc_vel = malloc(loc_n*sizeof(vect_t));
   if (my_rank == 0) vel =
         malloc(n*sizeof(vect_t));
   MPI_Type_contiguous(DIM, MPI_DOUBLE,
         &vect_mpi_t);
```

MPI Osnovna forma

```
   MPI_Type_commit(&vect_mpi_t);

   if (g_i == 'i')
      Get_init_cond(masses, pos, loc_vel, n,
         loc_n);
   else
      Gen_init_cond(masses, pos, loc_vel, n,
         loc_n);
```

MPI Osnovna forma

```
   start = MPI_Wtime();
#  ifndef NO_OUTPUT
   Output_state(0.0, masses, pos, loc_vel, n,
         loc_n);
#  endif
   for (step = 1; step <= n_steps; step++) {
      t = step*delta_t;
      for (loc_part = 0; loc_part < loc_n;
         loc_part++)
```

MPI Osnovna forma

```
         Compute_force(loc_part, masses,
         loc_forces, pos, n, loc_n);
      for (loc_part = 0; loc_part < loc_n;
         loc_part++)
         Update_part(loc_part, masses, loc_forces,
         loc_pos, loc_vel,
               n, loc_n, delta_t);
      MPI_Allgather(MPI_IN_PLACE, loc_n,
         vect_mpi_t,
                  pos, loc_n, vect_mpi_t, comm);
#     ifndef NO_OUTPUT
```

MPI Osnovna forma

```
      if (step % output_freq == 0)
         Output_state(t, masses, pos, loc_vel, n,
           loc_n);
#     endif
   }

   finish = MPI_Wtime();
   if (my_rank == 0)
```

MPI Osnovna forma

```
      printf("Elapsed time = %e seconds\n",
        finish-start);

   MPI_Type_free(&vect_mpi_t);
   free(masses);
   free(pos);
   free(loc_forces);
   free(loc_vel);
```

MPI Osnovna forma

```
   if (my_rank == 0) free(vel);

   MPI_Finalize();

   return 0;
}  /* main */
```

MPI Osnovna forma

```
/*---------------------------------------------------
        ---------------------
 * Function: Usage
 * Purpose:  Print instructions for command-line
       and exit
 * In arg:
 *    prog_name:  the name of the program as typed
       on the command-line
 */
```

MPI Osnovna forma

```
void Usage(char* prog_name) {

   fprintf(stderr, "usage: mpiexec -n <number of
        processes> %s\n", prog_name);
   fprintf(stderr, "   <number of particles>
        <number of timesteps>\n");
   fprintf(stderr, "   <size of timestep> <output
        frequency>\n");
   fprintf(stderr, "   <g|i>\n");
   fprintf(stderr, "   'g': program should
        generate init conds\n");
```

MPI Osnovna forma

```
   fprintf(stderr, "   'i': program should get
        init conds from stdin\n");

   exit(0);
}  /* Usage */


/*---------------------------------------------
        ---------------------
```

MPI Osnovna forma

```
* Function:  Get_args
* Purpose:   Get command line args
* In args:
*    argc:           number of command line args
*    argv:           command line args
* Out args:
*    n_p:            pointer to n, the number of
       particles
```

MPI Osnovna forma

```
*    n_steps_p:       pointer to n_steps, the
       number of timesteps
*    delta_t_p:       pointer to delta_t, the
       size of each timestep
*    output_freq_p:   pointer to output_freq,
       which is the number of
*                     timesteps between steps
       whose output is printed
*    g_i_p:           pointer to char which is
       'g' if the init conds
*                     should be generated by the
       program and 'i' if
*                     they should be read from
       stdin
```

MPI Osnovna forma

```
 */
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p) {
   if (my_rank == 0) {
      if (argc != 6) Usage(argv[0]);
      *n_p = strtol(argv[1], NULL, 10);
      *n_steps_p = strtol(argv[2], NULL, 10);
```

MPI Osnovna forma

```
      *delta_t_p = strtod(argv[3], NULL);
      *output_freq_p = strtol(argv[4], NULL, 10);
      *g_i_p = argv[5][0];
   }
   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
   MPI_Bcast(n_steps_p, 1, MPI_INT, 0, comm);
   MPI_Bcast(delta_t_p, 1, MPI_DOUBLE, 0, comm);
```

MPI Osnovna forma

```
   MPI_Bcast(output_freq_p, 1, MPI_INT, 0, comm);
   MPI_Bcast(g_i_p, 1, MPI_CHAR, 0, comm);

   if (*n_p <= 0 || *n_steps_p < 0 || *delta_t_p
        <= 0) {
      if (my_rank == 0) Usage(argv[0]);
      MPI_Finalize();
      exit(0);
```

MPI Osnovna forma

```
   }
   if (*g_i_p != 'g' && *g_i_p != 'i') {
      if (my_rank == 0) Usage(argv[0]);
      MPI_Finalize();
      exit(0);
   }
#  ifdef DEBUG
```

MPI Osnovna forma

```
   if (my_rank == 0) {
      printf("n = %d\n", *n_p);
      printf("n_steps = %d\n", *n_steps_p);
      printf("delta_t = %e\n", *delta_t_p);
      printf("output_freq = %d\n",
         *output_freq_p);
      printf("g_i = %c\n", *g_i_p);
   }
```

MPI Osnovna forma

```
#  endif
}  /* Get_args */


/*-------------------------------------------------
      ---------------------
 * Function:    Get_init_cond
 * Purpose:     Read in initial conditions:  mass,
       position and velocity
```

MPI Osnovna forma

```
 *             for each particle
 * In args:
 *    n:       total number of particles
 *    loc_n:   number of particles assigned to
       this process
 * Out args:
 *    masses:  global array of the masses of the
       particles
 *    pos:     global array of positions
```

MPI Osnovna forma

```
 *    loc_vel: local array of velocities assigned
       to this process.
 *
 * Global var:
 *    vel:     Scratch.  Used by process 0 for
       global velocities
 */
void Get_init_cond(double masses[], vect_t pos[],
     vect_t loc_vel[], int n, int loc_n) {
```

MPI Osnovna forma

```
   int part;

   if (my_rank == 0) {
      printf("For each particle, enter (in
        order):\n");
      printf("   its mass, its x-coord, its y-
        coord, ");
      printf("its x-velocity, its y-velocity\n");
      for (part = 0; part < n; part++) {
```

MPI Osnovna forma

```
         scanf("%lf", &masses[part]);
         scanf("%lf", &pos[part][X]);
         scanf("%lf", &pos[part][Y]);
         scanf("%lf", &vel[part][X]);
         scanf("%lf", &vel[part][Y]);
      }
   }
```

MPI Osnovna forma

```
   MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
   MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
   MPI_Scatter(vel, loc_n, vect_mpi_t,
         loc_vel, loc_n, vect_mpi_t, 0, comm);
}  /* Get_init_cond */

/*-------------------------------------------------
        ---------------------
```

MPI Osnovna forma

```
* Function:  Gen_init_cond
* Purpose:   Generate initial conditions:  mass,
      position and velocity
*           for each particle
* In args:
*    n:       total number of particles
*    loc_n:   number of particles assigned to
       this process
* Out args:
```

MPI Osnovna forma

```
*    masses:  global array of the masses of the
       particles
*    pos:     global array of positions
*    loc_vel: local array of velocities assigned
       to this process.
* Global var:
*    vel:     Scratch.  Used by process 0 for
       global velocities
*
* Note:      The initial conditions place all
       particles at
```

MPI Osnovna forma

```
 *            equal intervals on the nonnegative
       x-axis with
 *            identical masses, and identical
       initial speeds
 *            parallel to the y-axis.  However,
       some of the
 *            velocities are in the positive y-
       direction and
 *            some are negative.
 */
void Gen_init_cond(double masses[], vect_t pos[],
```

MPI Osnovna forma

```
      vect_t loc_vel[], int n, int loc_n) {
   int part;
   double mass = 5.0e24;
   double gap = 1.0e5;
   double speed = 3.0e4;

   if (my_rank == 0) {
```

MPI Osnovna forma

```
//    srandom(1);
      for (part = 0; part < n; part++) {
         masses[part] = mass;
         pos[part][X] = part*gap;
         pos[part][Y] = 0.0;
         vel[part][X] = 0.0;
//       if (random()/((double) RAND_MAX) >= 0.5)
```

MPI Osnovna forma

```
         if (part % 2 == 0)
            vel[part][Y] = speed;
         else
            vel[part][Y] = -speed;
      }
   }
```

MPI Osnovna forma

```
   MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
   MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
   MPI_Scatter(vel, loc_n, vect_mpi_t,
         loc_vel, loc_n, vect_mpi_t, 0, comm);
}  /* Gen_init_cond */
```

MPI Osnovna forma

```
/*---------------------------------------------
        ---------------------
 * Function:   Output_state
 * Purpose:    Print the current state of the
       system
 * In args:
 *    time:    current time
 *    masses:  global array of particle masses
 *    pos:     global array of particle positions
```

MPI Osnovna forma

```
 *    loc_vel: local array of my particle
         velocities
 *    n:       total number of particles
 *    loc_n:   number of my particles
 */
void Output_state(double time, double masses[],
        vect_t pos[],
      vect_t loc_vel[], int n, int loc_n) {
   int part;
```

MPI Osnovna forma

```
   MPI_Gather(loc_vel, loc_n, vect_mpi_t, vel,
         loc_n, vect_mpi_t,
         0, comm);
   if (my_rank == 0) {
      printf("%.2f\n", time);
      for (part = 0; part < n; part++) {
//       printf("%.3f ", masses[part]);
```

MPI Osnovna forma

```
         printf("%3d %10.3e ", part, pos[part]
         [X]);
         printf("  %10.3e ", pos[part][Y]);
         printf("  %10.3e ", vel[part][X]);
         printf("  %10.3e\n", vel[part][Y]);
      }
      printf("\n");
   }
```

MPI Osnovna forma

```
} /* Output_state */


/*--------------------------------------------------
      ---------------------
 * Function:       Compute_force
 * Purpose:        Compute the total force on
       particle loc_part.  Don't
 *                 exploit the symmetry (force on
       particle i due to
```

MPI Osnovna forma

```
*                particle k) = -(force on
      particle k due to particle i)
* In args:
*    loc_part:    the particle (local index) on
      which we're computing
*                the total force
*    masses:      global array of particle masses
*    pos:         global array of particle
      positions
*    n:           total number of particles
```

MPI Osnovna forma

```
 *    loc_n:       number of my particles
 * Out arg:
 *    loc_forces:  array of total forces acting on
        my particles
 *
 * Note: This function uses the force due to
        gravitation.  So
 * the force on particle i due to particle k is
        given by
 *
```

MPI Osnovna forma

```
 *    m_i m_k (s_k - s_i)/|s_k - s_i|^2
 *
 * Here, m_k is the mass of particle k and s_k is
        its position vector
 * (at time t).
 */
void Compute_force(int loc_part, double masses[],
        vect_t loc_forces[],
      vect_t pos[], int n, int loc_n) {
```

MPI Osnovna forma

```
   int k, part;
   double mg;
   vect_t f_part_k;
   double len, len_3, fact;

   /* Global index corresponding to loc_part */
   part = my_rank*loc_n + loc_part;
```

MPI Osnovna forma

```
   loc_forces[loc_part][X] = loc_forces[loc_part]
        [Y] = 0.0;
#  ifdef DEBUG
   printf("Proc %d > Current total force on part
        %d = (%.3e, %.3e)\n",
         my_rank, part, loc_forces[loc_part][X],
         loc_forces[loc_part][Y]);
#  endif
   for (k = 0; k < n; k++) {
```

MPI Osnovna forma

```
      if (k != part) {
         /* Compute force on part due to k */
         f_part_k[X] = pos[part][X] - pos[k][X];
         f_part_k[Y] = pos[part][Y] - pos[k][Y];
         len = sqrt(f_part_k[X]*f_part_k[X] +
        f_part_k[Y]*f_part_k[Y]);
         len_3 = len*len*len;
         mg = -G*masses[part]*masses[k];
```

MPI Osnovna forma

```
         fact = mg/len_3;
         f_part_k[X] *= fact;
         f_part_k[Y] *= fact;
#        ifdef DEBUG
         printf("Proc %d > Force on part %d due to
        part %d = (%.3e, %.3e)\n",
               my_rank, part, k, f_part_k[X],
        f_part_k[Y]);
#        endif
```

MPI Osnovna forma

```
         /* Add force in to total forces */
         loc_forces[loc_part][X] += f_part_k[X];
         loc_forces[loc_part][Y] += f_part_k[Y];
      }
   }
}  /* Compute_force */
```

MPI Osnovna forma

```
/*--------------------------------------------------
        ---------------------
 * Function:  Update_part
 * Purpose:   Update the velocity and position for
       particle loc_part
 * In args:
 *    loc_part:    local index of the particle
       we're updating
```

MPI Osnovna forma

```
 *    masses:      global array of particle masses
 *    loc_forces:  local array of total forces
 *    n:           total number of particles
 *    loc_n:       number of particles assigned to
        this process
 *    delta_t:     step size
 *
 * In/out args:
```

MPI Osnovna forma

```
 *    loc_pos:     local array of positions
 *    loc_vel:     local array of velocities
 *
 * Note:  This version uses Euler's method to
          update both the velocity
 *    and the position.
 */
void Update_part(int loc_part, double masses[],
        vect_t loc_forces[],
```

MPI Osnovna forma

```
      vect_t loc_pos[], vect_t loc_vel[], int n,
        int loc_n,
      double delta_t) {
   int part;
   double fact;

   part = my_rank*loc_n + loc_part;
   fact = delta_t/masses[part];
```

MPI Osnovna forma

```
#  ifdef DEBUG
   printf("Proc %d > Before update of %d:\n",
         my_rank, part);
   printf("   Position  = (%.3e, %.3e)\n",
         loc_pos[loc_part][X], loc_pos[loc_part]
         [Y]);
   printf("   Velocity  = (%.3e, %.3e)\n",
         loc_vel[loc_part][X], loc_vel[loc_part]
         [Y]);
   printf("   Net force = (%.3e, %.3e)\n",
```

MPI Osnovna forma

```
            loc_forces[loc_part][X],
            loc_forces[loc_part][Y]);
#  endif
   loc_pos[loc_part][X] += delta_t *
         loc_vel[loc_part][X];
   loc_pos[loc_part][Y] += delta_t *
         loc_vel[loc_part][Y];
   loc_vel[loc_part][X] += fact *
         loc_forces[loc_part][X];
   loc_vel[loc_part][Y] += fact *
         loc_forces[loc_part][Y];
#  ifdef DEBUG
```

MPI Osnovna forma

```
   printf("Proc %d > Position of %d = (%.3e,
         %.3e), Velocity = (%.3e,%.3e)\n",
          my_rank, part, loc_pos[loc_part][X],
          loc_pos[loc_part][Y],
                loc_vel[loc_part][X],
          loc_vel[loc_part][Y]);
#  endif
}  /* Update_part */
```

MPI Redukovana forma

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>

#define DIM 2  /* Two-dimensional system */
```

MPI Redukovana forma

```
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */

typedef double vect_t[DIM];  /* Vector type for
        position, etc. */

/* Global variables.  Except for vel all are
        unchanged after being set */
const double G = 6.673e-11;  /* Gravitational
        constant. */
```

MPI Redukovana forma

```
                            /* Units are
        m^3/(kg*s^2)  */
int my_rank, comm_sz;
MPI_Comm comm;
MPI_Datatype vect_mpi_t;
MPI_Datatype cyclic_mpi_t;

/* Scratch arrays used by process 0 for I/O */
```

MPI Redukovana forma

```
vect_t *vel = NULL;
vect_t *pos = NULL;

void Usage(char* prog_name);
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p);
void Build_cyclic_mpi_type(int loc_n);
```

MPI Redukovana forma

```
void Get_init_cond(double masses[], vect_t
        loc_pos[],
      vect_t loc_vel[], int n, int loc_n);
void Gen_init_cond(double masses[], vect_t
        loc_pos[],
      vect_t loc_vel[], int n, int loc_n);
void Output_state(double time, double masses[],
        vect_t loc_pos[],
      vect_t loc_vel[], int n, int loc_n);
void Compute_forces(double masses[], vect_t
        tmp_data[],
```

MPI Redukovana forma

```
      vect_t loc_forces[], vect_t loc_pos[], int
        n, int loc_n);
void Compute_proc_forces(double masses[], vect_t
        tmp_data[],
      vect_t loc_forces[], vect_t pos1[], int
        loc_n1, int rk1,
      int loc_n2, int rk2, int n, int p);
int Local_to_global(int loc_part, int proc_rk, int
        proc_count);
int Global_to_local(int gbl_part, int proc_rk, int
        proc_count);
int First_index(int gbl1, int proc_rk1, int
        proc_rk2, int proc_count);
```

MPI Redukovana forma

```
void Compute_force_pair(double m1, double m2,
      vect_t pos1, vect_t pos2,
     vect_t force1, vect_t force2);
void Update_part(int loc_part, double masses[],
      vect_t loc_forces[],
     vect_t loc_pos[], vect_t loc_vel[], int n,
      int loc_n, double delta_t);

/*--------------------------------------------------
      --------------------*/
int main(int argc, char* argv[]) {
```

MPI Redukovana forma

```
   int n;                    /* Total number of
        particles     */
   int loc_n;                /* Number of my
        particles        */
   int n_steps;              /* Number of
        timesteps           */
   int step;                 /* Current step
        */
   int output_freq;          /* Frequency of
        output           */
   double delta_t;           /* Size of timestep
        */
   double t;                 /* Current Time
        */
```

MPI Redukovana forma

```
   double* masses;           /* All the masses
        */
   vect_t* loc_pos;          /* Positions of my
        particles     */
   vect_t* tmp_data;         /* Received
        positions and forces */
   vect_t* loc_vel;          /* Velocities of my
        particles    */
   vect_t* loc_forces;       /* Forces on my
        particles        */
   int loc_part;
```

MPI Redukovana forma

```
   char g_i;                 /*_G_en or _i_nput
        init conds */
   double start, finish;     /* For timings
        */

   MPI_Init(&argc, &argv);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &comm_sz);
   MPI_Comm_rank(comm, &my_rank);
```

MPI Redukovana forma

```
Get_args(argc, argv, &n, &n_steps, &delta_t,
      &output_freq, &g_i);
loc_n = n/comm_sz;  /* n should be evenly
      divisible by comm_sz */
masses = malloc(n*sizeof(double));
tmp_data = malloc(2*loc_n*sizeof(vect_t));
loc_forces = malloc(loc_n*sizeof(vect_t));
loc_pos = malloc(loc_n*sizeof(vect_t));
```

MPI Redukovana forma

```
   loc_vel = malloc(loc_n*sizeof(vect_t));
   if (my_rank == 0) {
      pos = malloc(n*sizeof(vect_t));
      vel = malloc(n*sizeof(vect_t));
   }
   MPI_Type_contiguous(DIM, MPI_DOUBLE,
         &vect_mpi_t);
   MPI_Type_commit(&vect_mpi_t);
```

MPI Redukovana forma

```
   Build_cyclic_mpi_type(loc_n);

   if (g_i == 'i')
      Get_init_cond(masses, loc_pos, loc_vel, n,
         loc_n);
   else
      Gen_init_cond(masses, loc_pos, loc_vel, n,
         loc_n);
```

MPI Redukovana forma

```
   start = MPI_Wtime();
#  ifndef NO_OUTPUT
   Output_state(0.0, masses, loc_pos, loc_vel, n,
         loc_n);
#  endif
   for (step = 1; step <= n_steps; step++) {
      t = step*delta_t;
      Compute_forces(masses, tmp_data, loc_forces,
         loc_pos,
```

MPI Redukovana forma

```
            n, loc_n);
      for (loc_part = 0; loc_part < loc_n;
         loc_part++)
          Update_part(loc_part, masses, loc_forces,
         loc_pos, loc_vel,
               n, loc_n, delta_t);
#     ifndef NO_OUTPUT
      if (step % output_freq == 0)
          Output_state(t, masses, loc_pos, loc_vel,
         n, loc_n);
```

MPI Redukovana forma

```
#     endif
   }

   finish = MPI_Wtime();
   if (my_rank == 0)
      printf("Elapsed time = %e seconds\n",
         finish-start);
```

MPI Redukovana forma

```
   MPI_Type_free(&vect_mpi_t);
   MPI_Type_free(&cyclic_mpi_t);
   free(masses);
   free(tmp_data);
   free(loc_forces);
   free(loc_pos);
   free(loc_vel);
```

MPI Redukovana forma

```
   if (my_rank == 0) {
      free(pos);
      free(vel);
   }

   MPI_Finalize();
```

MPI Redukovana forma

```
   return 0;
}  /* main */


/*-------------------------------------------------
      ---------------------
 * Function: Usage
 * Purpose:  Print instructions for command-line
      and exit
```

MPI Redukovana forma

```
 * In arg:
 *    prog_name:  the name of the program as typed
       on the command-line
 */
void Usage(char* prog_name) {

   fprintf(stderr, "usage: mpiexec -n <number of
       processes> %s\n", prog_name);
   fprintf(stderr, "   <number of particles>
       <number of timesteps>\n");
```

MPI Redukovana forma

```
   fprintf(stderr, "   <size of timestep> <output
       frequency>\n");
   fprintf(stderr, "   <g|i>\n");
   fprintf(stderr, "   'g': program should
       generate init conds\n");
   fprintf(stderr, "   'i': program should get
       init conds from stdin\n");

}  /* Usage */
```

MPI Redukovana forma

```


/*-------------------------------------------------
      ---------------------
 * Function:  Get_args
 * Purpose:   Get command line args
 * In args:
 *    argc:            number of command line args
 *    argv:            command line args
```

MPI Redukovana forma

```
* Out args:
*    n_p:             pointer to n, the number of
      particles
*    n_steps_p:       pointer to n_steps, the
      number of timesteps
*    delta_t_p:       pointer to delta_t, the
      size of each timestep
*    output_freq_p:   pointer to output_freq,
      which is the number of
*                     timesteps between steps
      whose output is printed
*    g_i_p:           pointer to char which is
      'g' if the init conds
```

MPI Redukovana forma

```
 *                    should be generated by the
       program and 'i' if
 *                    they should be read from
       stdin
 */
void Get_args(int argc, char* argv[], int* n_p,
        int* n_steps_p,
      double* delta_t_p, int* output_freq_p, char*
        g_i_p) {
   if (my_rank == 0) {
      if (argc != 6) {
```

MPI Redukovana forma

```
         Usage(argv[0]);
         *n_p = *n_steps_p = *output_freq_p = 0;
         *delta_t_p = 0.0;
         *g_i_p = 'g';
      } else {
         *n_p = strtol(argv[1], NULL, 10);
         *n_steps_p = strtol(argv[2], NULL, 10);
```

MPI Redukovana forma

```
         *delta_t_p = strtod(argv[3], NULL);
         *output_freq_p = strtol(argv[4], NULL,
        10);
         *g_i_p = argv[5][0];
      }
   }
   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
   MPI_Bcast(n_steps_p, 1, MPI_INT, 0, comm);
```

MPI Redukovana forma

```
   MPI_Bcast(delta_t_p, 1, MPI_DOUBLE, 0, comm);
   MPI_Bcast(output_freq_p, 1, MPI_INT, 0, comm);
   MPI_Bcast(g_i_p, 1, MPI_CHAR, 0, comm);

   if (*n_p <= 0 || *n_steps_p < 0 || *delta_t_p
         <= 0) {
      if (my_rank == 0 && argc > 6)
         Usage(argv[0]);
      MPI_Finalize();
```

MPI Redukovana forma

```
      exit(0);
   }
   if (*g_i_p != 'g' && *g_i_p != 'i') {
      if (my_rank == 0) Usage(argv[0]);
      MPI_Finalize();
      exit(0);
   }
```

MPI Redukovana forma

```
#  ifdef DEBUG
   if (my_rank == 0) {
      printf("n = %d\n", *n_p);
      printf("n_steps = %d\n", *n_steps_p);
      printf("delta_t = %e\n", *delta_t_p);
      printf("output_freq = %d\n",
        *output_freq_p);
      printf("g_i = %c\n", *g_i_p);
```

MPI Redukovana forma

```
   }
#  endif
}  /* Get_args */

/*-------------------------------------------------
        --------------------
 * Function:         Build_cyclic_mpi_type
 * Purpose:          Build an MPI derived datatype
       that can be used with
```

MPI Redukovana forma

```
 *                    cyclically distributed data.
 * In arg:
 *    loc_n:          The number of elements
 *       assigned to each process
 * Global out:
 *    cyclic_mpi_t:   An MPI datatype that can be
 *       used with cyclically
 *                    distributed data
 */
```

MPI Redukovana forma

```
void Build_cyclic_mpi_type(int loc_n) {
   MPI_Datatype temp_mpi_t;
   MPI_Aint lb, extent;

   MPI_Type_vector(loc_n, 1, comm_sz, vect_mpi_t,
        &temp_mpi_t);
   MPI_Type_get_extent(vect_mpi_t, &lb, &extent);
   MPI_Type_create_resized(temp_mpi_t, lb, extent,
        &cyclic_mpi_t);
```

MPI Redukovana forma

```
   MPI_Type_commit(&cyclic_mpi_t);

}  /* Build_cyclic_mpi_type */


/*---------------------------------------------
       ---------------------
 * Function:   Get_init_cond
```

MPI Redukovana forma

```
 * Purpose:    Read in initial conditions:  mass,
       position and velocity
 *             for each particle
 * In args:
 *    n:       total number of particles
 *    loc_n:   number of particles assigned to
       this process
 * Out args:
 *    masses:  global array of the masses of the
       particles
```

MPI Redukovana forma

```
*    loc_pos: local array of the positions of the
       particles assigned
*             to this process
*    loc_vel: local array of velocities assigned
       to this process.
*
* Global var:
*    pos:     Scratch.  Used by process 0 for
       global positions
*    vel:     Scratch.  Used by process 0 for
       global velocities
```

MPI Redukovana forma

```
 */
void Get_init_cond(double masses[], vect_t
        loc_pos[],
      vect_t loc_vel[], int n, int loc_n) {
   int part;

   if (my_rank == 0) {
      printf("For each particle, enter (in
        order):\n");
```

MPI Redukovana forma

```
      printf("   its mass, its x-coord, its y-
        coord, ");
      printf("its x-velocity, its y-velocity\n");
      for (part = 0; part < n; part++) {
         scanf("%lf", &masses[part]);
         scanf("%lf", &pos[part][X]);
         scanf("%lf", &pos[part][Y]);
         scanf("%lf", &vel[part][X]);
```

MPI Redukovana forma

```
         scanf("%lf", &vel[part][Y]);
      }
   }
   MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
   MPI_Scatter(pos, 1, cyclic_mpi_t,
         loc_pos, loc_n, vect_mpi_t, 0, comm);
   MPI_Scatter(vel, 1, cyclic_mpi_t,
```

MPI Redukovana forma

```
        loc_vel, loc_n, vect_mpi_t, 0, comm);
}  /* Get_init_cond */


/*--------------------------------------------------
       ---------------------
 * Function:  Gen_init_cond
 * Purpose:   Generate initial conditions:  mass,
       position and velocity
```

MPI Redukovana forma

```
*           for each particle
* In args:
*    n:       total number of particles
*    loc_n:   number of particles assigned to
       this process
* Out args:
*    masses:  global array of the masses of the
       particles
*    pos:     global array of positions
```

MPI Redukovana forma

```
*    loc_pos: local array of the positions of the
       particles assigned
*             to this process
*    loc_vel: local array of velocities assigned
       to this process.
* Global vars:
*    pos:     Scratch.  Used by process 0 for
       global positions
*    vel:     Scratch.  Used by process 0 for
       global velocities
*
```

MPI Redukovana forma

```
* Note:      The initial conditions place all
       particles at
*           equal intervals on the nonnegative
       x-axis with
*           identical masses, and identical
       initial speeds
*           parallel to the y-axis.  However,
       some of the
*           velocities are in the positive y-
       direction and
*           some are negative.
*/
```

MPI Redukovana forma

```
void Gen_init_cond(double masses[], vect_t
        loc_pos[],
      vect_t loc_vel[], int n, int loc_n) {
   int part;
   double mass = 5.0e24;
   double gap = 1.0e5;
   double speed = 3.0e4;
```

MPI Redukovana forma

```
   if (my_rank == 0) {
//    srandom(1);
      for (part = 0; part < n; part++) {
         masses[part] = mass;
         pos[part][X] = part*gap;
         pos[part][Y] = 0.0;
         vel[part][X] = 0.0;
```

MPI Redukovana forma

```
//       if (random()/((double) RAND_MAX) >= 0.5)
         if (part % 2 == 0)
            vel[part][Y] = speed;
         else
            vel[part][Y] = -speed;
      }
   }
```

MPI Redukovana forma

```
   MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
   MPI_Scatter(pos, 1, cyclic_mpi_t,
         loc_pos, loc_n, vect_mpi_t, 0, comm);
   MPI_Scatter(vel, 1, cyclic_mpi_t,
         loc_vel, loc_n, vect_mpi_t, 0, comm);
}  /* Gen_init_cond */
```

MPI Redukovana forma

```
/*--------------------------------------------------
        ---------------------
 * Function:   Output_state
 * Purpose:    Print the current state of the
       system
 * In args:
 *    time:    current time
```

MPI Redukovana forma

```
 *    masses:  global array of particle masses
 *    loc_pos: local array of particle positions
 *    loc_vel: local array of my particle
       velocities
 *    n:       total number of particles
 *    loc_n:   number of my particles
 * Global vars:
 *    pos:     Scratch.  Used by proc 0 for global
       positions
```

MPI Redukovana forma

```
 *    vel:     Scratch.  Used by proc 0 for global
       velocities
 */
void Output_state(double time, double masses[],
        vect_t loc_pos[],
      vect_t loc_vel[], int n, int loc_n) {
   int part;

   MPI_Gather(loc_pos, loc_n, vect_mpi_t, pos, 1,
         cyclic_mpi_t,
```

MPI Redukovana forma

```
         0, comm);
   MPI_Gather(loc_vel, loc_n, vect_mpi_t, vel, 1,
         cyclic_mpi_t,
         0, comm);
   if (my_rank == 0) {
      printf("%.2f\n", time);
      for (part = 0; part < n; part++) {
//       printf("%.3f ", masses[part]);
```

MPI Redukovana forma

```
            printf("%3d %10.3e ", part, pos[part]
          [X]);
            printf("  %10.3e ", pos[part][Y]);
            printf("  %10.3e ", vel[part][X]);
            printf("  %10.3e\n", vel[part][Y]);
        }
        printf("\n");
    }
```

MPI Redukovana forma

```
}  /* Output_state */


/*---------------------------------------------
          ---------------------
 * Function:       Compute_forces
 * Purpose:        Compute the total force on each
       local particle.
 *                 Exploit the symmetry (force on
       particle i due to
```

MPI Redukovana forma

```
 *                 particle k) = -(force on
       particle k due to particle i)
 * In args:
 *    masses:      global array of particle masses
       (dimension n)
 *    loc_pos:     local array of positions of my
       particles (dim loc_n)
 *    n:           total number of particles
 *    loc_n:       number of my particles
 * Scratch:
```

MPI Redukovana forma

```
 *    tmp_data:    Stores received positions
       (subscripts 0 - loc_n-1)
 *                 and forces computed thus far on
       particles corresp
 *                 onding to received positions
       (subscripts
 *                 loc_n - 2*loc_n-1).  (dimension
       2*loc_n)
 * Out arg:
 *    loc_forces:  array of total forces acting on
       my particles
 */
```

MPI Redukovana forma

```
void Compute_forces(double masses[], vect_t
        tmp_data[],
      vect_t loc_forces[], vect_t loc_pos[], int
        n, int loc_n) {
   int src, dest;  /* Source and dest processes
        for particle pos */
   int i, other_proc, loc_part;
   MPI_Status status;

   src = (my_rank + 1) % comm_sz;
```

MPI Redukovana forma

```
   dest = (my_rank - 1 + comm_sz) % comm_sz;
   memcpy(tmp_data, loc_pos,
        loc_n*sizeof(vect_t));
   memset(tmp_data + loc_n, 0,
        loc_n*sizeof(vect_t));
   memset(loc_forces, 0, loc_n*sizeof(vect_t));

   /* First compute the forces resulting from my
        particles' interactions
    * with themselves */
```

MPI Redukovana forma

```
   Compute_proc_forces(masses, tmp_data,
        loc_forces, loc_pos, loc_n,
         my_rank, loc_n, my_rank, n, comm_sz);
   /* Now compute forces resulting from my
        particles' interactions with
    * other processes' particles */
   for (i = 1; i < comm_sz; i++) {
      other_proc = (my_rank + i) % comm_sz;
      MPI_Sendrecv_replace(tmp_data, 2*loc_n,
         vect_mpi_t, dest, 0, src, 0,
```

MPI Redukovana forma

```
            comm, &status);
      Compute_proc_forces(masses, tmp_data,
         loc_forces, loc_pos, loc_n,
            my_rank, loc_n, other_proc, n,
         comm_sz);
   }
   MPI_Sendrecv_replace(tmp_data, 2*loc_n,
        vect_mpi_t, dest, 0, src, 0,
         comm, &status);
   for (loc_part = 0; loc_part < loc_n;
        loc_part++) {
```

MPI Redukovana forma

```
      loc_forces[loc_part][X] +=
        tmp_data[loc_n+loc_part][X];
      loc_forces[loc_part][Y] +=
        tmp_data[loc_n+loc_part][Y];
   }

}  /* Compute_forces */
```

MPI Redukovana forma

```
/*--------------------------------------------------
        ---------------------
 * Function:       Compute_proc_forces
 * Purpose:        Compute the forces on particles
       owned by process
 *                 rk1 due to interaction with
       particles owned by
 *                 procss rk2.  Exploit the
       symmetry (force on particle
 *                 i due to particle k) = -(force
       on particle k due
 *                 to particle i)
```

MPI Redukovana forma

```
 * In args:
 *    masses:      global array of particle masses
        (dim n)
 *    pos1:        local array of particle
        positions (dim loc_n1)
 *    loc_n1:      number of my particles in pos1
 *    rk1:         process owning particles in
        pos1
 *    loc_n2:      number of particles contributed
        by second process
 *    rk2:         process owning contributed
        particles
```

MPI Redukovana forma

```
 *    n:           total number of particles
 *    p:           number of processes in
        communicator containing
 *                 processes rk1 and rk2
 * In/out args:
 *    tmp_data:    positions of rk2 particles (in
        only, loc_n2 positions)
 *                 followed by forces computed
       thus far corresp to
 *                   rk2 particles (in and out,
       loc_n2 positions)
```

MPI Redukovana forma

```
 *    loc_forces:  forces computed thus far on my
        particles (loc_n1)
 */
void Compute_proc_forces(double masses[], vect_t
        tmp_data[],
      vect_t loc_forces[], vect_t pos1[], int
        loc_n1, int rk1,
      int loc_n2, int rk2, int n, int p) {
   int loc_part1, loc_part2;
   int gbl_part1, gbl_part2;
```

MPI Redukovana forma

```
   for (gbl_part1 = rk1, loc_part1 = 0;
        loc_part1 < loc_n1;
        loc_part1++, gbl_part1 += p) {
      for(gbl_part2 = First_index(gbl_part1, rk1,
        rk2, p),
          loc_part2 = Global_to_local(gbl_part2,
        rk2, p);
          loc_part2 < loc_n2;
```

MPI Redukovana forma

```
          loc_part2++, gbl_part2 += p) {
#        ifdef DEBUG
         printf("Proc %d > Current total force on
        part %d = (%.3e, %.3e)\n",
               my_rank, gbl_part1,
        loc_forces[loc_part1][X],
               loc_forces[loc_part1][Y]);
         printf("Proc %d > Current total force on
        part %d = (%.3e, %.3e)\n",
               my_rank, gbl_part2,
```

MPI Redukovana forma

```
               tmp_data[loc_n2+loc_part2][X],
               tmp_data[loc_n2+loc_part2][Y]);
#        endif
         Compute_force_pair(masses[gbl_part1],
        masses[gbl_part2],
               pos1[loc_part1],
        tmp_data[loc_part2],
               loc_forces[loc_part1],
        tmp_data[loc_n2+loc_part2]);
#        ifdef DEBUG
```

MPI Redukovana forma

```
         printf("Proc %d > Current total force on
        part %d = (%.3e, %.3e)\n",
               my_rank, gbl_part1,
        loc_forces[loc_part1][X],
               loc_forces[loc_part1][Y]);
         printf("Proc %d > Current total force on
        part %d = (%.3e, %.3e)\n",
               my_rank, gbl_part2,
               tmp_data[loc_n2+loc_part2][X],
               tmp_data[loc_n2+loc_part2][Y]);
```

MPI Redukovana forma

```
#        endif
      } /* for gbl_part2 */
   } /* for gbl_part1 */
}  /* Compute_proc_forces */

/*--------------------------------------------------
         ---------------------
 * Function:    Local_to_global
```

MPI Redukovana forma

```
* Purpose:     Convert a local particle index to
       a global particle
*             index
* In args:
*    loc_part:  local particle index
*    proc_rk:   process rank
*    loc_n:     number of particles assigned to
       the process
*    n:         total number of particles
```

MPI Redukovana forma

```
*    proc_count:  number of processes in the
       communicator
* Ret val:
*    global particle index
*
* Notes:
* 1.  This version assumes a cyclic distribution
       of the particles
* 2.  It also assumes loc_n = n/proc_count, and n
       is evenly divisible
```

MPI Redukovana forma

```
 *     by proc_count
 */
int Local_to_global(int loc_part, int proc_rk, int
        proc_count) {
   return loc_part*proc_count + proc_rk;
}  /*  Local_to_global */

/*-----------------------------------------------
        ---------------------
```

MPI Redukovana forma

```
 * Function:    Global_to_local
 * Purpose:     Convert a global particle index to
       a global permuted
 *              index
 * In args:
 *    gbl_part:    The global particle index
 *    proc_rk:     The rank of the owning process
 *    proc_count:  The number of processes
```

MPI Redukovana forma

```
 *
 * Notes:
 * 1.  This version assumes a cyclic distribution
       of the particles
 * 2.  It also assumes loc_n = n/proc_count, and n
       is evenly divisible
 *     by proc_count
 */
int Global_to_local(int gbl_part, int proc_rk, int
        proc_count) {
```

MPI Redukovana forma

```
   return (gbl_part - proc_rk)/proc_count;
}  /* Global_to_local */


/*-----------------------------------------------
        ---------------------
 * Function:           First_index
 * Purpose:            Given a global index glb1
       assigned to process
```

MPI Redukovana forma

```
 *                    rk1, find the next higher
       global index assigned
 *                    to process rk2
 * In args:
 *    gbl1:           global particle index of
       particle assigned to
 *                    process rk1
 *    rk1:            rank of process owning gbl1
 *    rk2:            rank of process owning
       particle with computed index
```

MPI Redukovana forma

```
 *    proc_count:     number of processes
 * Return val:        next higher global particle
       index of particle assigned
 *                    to process rk2
 * Note:              If there is no particle
       assigned to rk2 with index
 *                    greater than rk1, the
       function will return a value
 *                    larger than n, the total
       number of particles.
 */
```

MPI Redukovana forma

```
int First_index(int gbl1, int rk1, int rk2, int
       proc_count) {
   if (rk1 < rk2)
      return gbl1 + (rk2 - rk1);
   else
      return gbl1 + (rk2 - rk1) + proc_count;
}  /* First_index */
```

MPI Redukovana forma

```
/*--------------------------------------------------
       ---------------------
 * Function:          Compute_force_pair
 * Purpose:           Compute the force resulting
       from the interaction of
 *                    of two particles.  Exploit
       the fact that f_kq = -f_qk
 * In args:
 *    m1, m2:         Masses of the two particles
```

MPI Redukovana forma

```
 *    pos1, pos2:      Positions of the two
        particles
 * In/out args:
 *    force1, force2:  The total forces on the two
        particles as thus far
 *                     computed
 */
void Compute_force_pair(double m1, double m2,
        vect_t pos1, vect_t pos2,
      vect_t force1, vect_t force2) {
```

MPI Redukovana forma

```
   double mg;
   vect_t f_part_k;
   double len, len_3, fact;

   f_part_k[X] = pos1[X] - pos2[X];
   f_part_k[Y] = pos1[Y] - pos2[Y];
   len = sqrt(f_part_k[X]*f_part_k[X] +
        f_part_k[Y]*f_part_k[Y]);
```

MPI Redukovana forma

```
   len_3 = len*len*len;
   mg = -G*m1*m2;
   fact = mg/len_3;
   f_part_k[X] *= fact;
   f_part_k[Y] *= fact;

   /* Add force in to total forces */
```

MPI Redukovana forma

```
   force1[X] += f_part_k[X];
   force1[Y] += f_part_k[Y];
   force2[X] -= f_part_k[X];
   force2[Y] -= f_part_k[Y];
}  /* Compute_force_pair */

/*-------------------------------------------------
        ---------------------
```

MPI Redukovana forma

```
 * Function:  Update_part
 * Purpose:   Update the velocity and position for
       particle loc_part
 * In args:
 *    loc_part:    local index of the particle
       we're updating
 *    masses:      global array of particle masses
 *    loc_forces:  local array of total forces
 *    n:           total number of particles
```

MPI Redukovana forma

```
 *    loc_n:       number of particles assigned to
       this process
 *    delta_t:     step size
 *
 * In/out args:
 *    loc_pos:     local array of positions
 *    loc_vel:     local array of velocities
 *
```

MPI Redukovana forma

```
 * Note:  This version uses Euler's method to
       update both the velocity
 *    and the position.
 */
void Update_part(int loc_part, double masses[],
        vect_t loc_forces[],
      vect_t loc_pos[], vect_t loc_vel[], int n,
        int loc_n,
      double delta_t) {
   int part;
```

MPI Redukovana forma

```
   double fact;

   part = my_rank*loc_n + loc_part;
   fact = delta_t/masses[part];
#  ifdef DEBUG
   printf("Proc %d > Before update of %d:\n",
        my_rank, part);
   printf("   Position  = (%.3e, %.3e)\n",
```

MPI Redukovana forma

```
          loc_pos[loc_part][X], loc_pos[loc_part]
        [Y]);
    printf("   Velocity  = (%.3e, %.3e)\n",
          loc_vel[loc_part][X], loc_vel[loc_part]
        [Y]);
    printf("   Net force = (%.3e, %.3e)\n",
          loc_forces[loc_part][X],
        loc_forces[loc_part][Y]);
#   endif
    loc_pos[loc_part][X] += delta_t *
        loc_vel[loc_part][X];
```

MPI Redukovana forma

```
    loc_pos[loc_part][Y] += delta_t *
        loc_vel[loc_part][Y];
    loc_vel[loc_part][X] += fact *
        loc_forces[loc_part][X];
    loc_vel[loc_part][Y] += fact *
        loc_forces[loc_part][Y];
#   ifdef DEBUG
    printf("Proc %d > Position of %d = (%.3e,
        %.3e), Velocity = (%.3e,%.3e)\n",
          my_rank, part, loc_pos[loc_part][X],
        loc_pos[loc_part][Y],
              loc_vel[loc_part][X],
        loc_vel[loc_part][Y]);
# endif
}
```