MPI – Message Passing Interface

Charakteristika

- Původně pro C a Fortran, dnes třeba i mpiJava
- Komunikační rozhraní pro paralelní programy
- Knihovna s definovaným API
  - Nezávislé implementace, možnost přenositelnosti
  - Možnost optimalizace pro různý hardware
- Častým cílovým prostředím MPI aplikací jsou clustery
  - Skupina počítačů propojených sítí, které se navenek tváří jako jeden superpočítač s distribuovanou pamětí
  - Počítače v clusteru spolupracují - komunikují pomocí jasně definovaného rozhraní
    - Změna jedné komponenty nevyžaduje změnu ostatních
  - Cílem je např. zvýšení výkonnosti, spolehlivosti…
  - Dalším prostředím jsou i paralelní počítače a gridy

- V současné době se používají dva standardy
  - MPI-1.2 se statickým běhovým prostředím
  - MPI-2.1 s novými vymoženostmi jako je paralelní I/O, dynamická správa procesů
  - MPI-1.2 je označováno jako legacy
    - I takové programy běží pod MPI-2.1
    - MPI-1.2 nabízí podmnožinu funkcí MP-2.1

- Místo vláken procesy, které tvoří distribuovanou aplikaci
- Snahou MPI je pokrýt aspekty meziprocesové komunikace

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

if (myid == 0) {
  printf("%d: We have %d processes\n",
              myid, numprocs);

  for (i=1;i<numprocs;i++) {
    sprintf(buff, "Hello %d! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i,
                    TAG, MPI_COMM_WORLD);
   }

   for (i=1;i<numprocs;i++) {
     MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
                          MPI_COMM_WORLD, &stat);
     printf("%d: %s\n", myid, buff);
    }
   } else {

     /* receive from rank 0: */
     MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0,
                    TAG, MPI_COMM_WORLD, &stat);
     sprintf(idstr, "Processor %d ", myid);
     strcat(buff, idstr);
     strcat(buff, "reporting for duty\n");
     /* send to rank 0: */
     MPI_Send(buff, BUFSIZE, MPI_CHAR, 0,
                             TAG, MPI_COMM_WORLD);
   }

   MPI_Finalize(); /* MPI Programs end with
                      MPI Finalize; this is a
                   weak synchronization point */
```

http://en.wikipedia.org/wiki/Message_Passing_Interface

Částečné porovnání MPI a PVM

- oba mohou běžet v heterogenním prostředí, ale musí to umožňovat jejich implementace a implementace aplikace
  - PVM však vzniklo v prostředí heterogenní sítě
  - MPI bylo navrženo spíše pro clustery – homogenní prostředí

- MPI vzniklo až po PVM a vychází z něj
  - MPI nabízí jednodušší a efektivnější abstrakce na vyšší úrovni, které umožňují uživateli definovat datové struktury a ty pak přenášet ve zprávách
    - vs. opaque data v XDR u PVM
  - MPI nabízí bohatší možnosti pro přenos zpráv

- PVM se nestará o topologii, MPI podporuje logické komunikační topologie
- MPI má už v návrhu snahu o omezení kopírování paměťových bloků – memory copies

- Programátor PVM může využít funkci pvm_config, aby například určil, kde spustí další proces – MPI nic takového nemá
  - Prý je informace okamžitě zastaralá, protože hned po jejím přečtení se může provést pvm_delhosts
    - Takto nestabilní prostředí by se ovšem vůbec nehodilo pro náročný paralelní výpočet

- PVM umožňuje programátorovi napojit se do systému pomocí nízko-úrovňových rutin PVM, MPI se tomu vyhýbá v rámci větší přenositelnosti
- http://www-unix.mcs.anl.gov/~gropp/bib/papers/2002/mpiandpvm.pdf

Jádro práce s MPI

- MPI_Init
  - Inicializace MPI
  - Proces deklaruje, že bude používat MPI

- MPI_Comm_Size
  - Vrací počet procesů používajících MPI
- MPI_Comm_Rank
  - Vrací vlastní identifikační číslo v MPI (viz dále)
  - ID se používá pro zasílání zpráv

- MPI_Send
  - Odeslání zprávy
- MPI_Recv
  - Příjem zprávy

- MPI_Finalize
  - Proces deklaruje, že už nebude používat MPI
  - Úklid a rozlučka s MPI

Skupiny, komunikátory a kontext

- Zasílání zpráv
  - rank příjemce
  - komunikátor
  - message tag

- message tag
  - rozlišují typy zpráv – skutečný obsah zprávy
  - dvě různé zprávy mohou mít stejný tag, pokud jsou použity v různých kontextech různých komunikátorů
  - střezte se dne, kdy použijete něčí knihovnu a ta bude používat vaše tagy => kontext

- skupina (group)
  - seznam procesů
  - proces je určen svým pořadím ve skupině
  - při hromadné komunikaci jsou příjemci adresováni skupinou
  - proces může být v několika skupinách
  - operace se skupinou je lokální operace

- komunikátor
  - vytváří se nad skupinou
  - popisuje skupinu procesů, má kontext, vyrovnávací paměť a obsahuje další specifické informace
  - analogie s radiovou frekvencí

  - intra-komunikátory
    - komunikace uvnitř skupiny
    - MPI_COMM_WORLD – všechny procesy v MPI aplikaci
    - lze mu přiřadit virtuální topologii
      - mřížka, nebo obecný graf
      - proces pak může komunikovat jenom se svými sousedy

- MPI_Cart_create(current_comm, dim_num, dim_sizes, wrap, reorder, &grid_comm);
  - vytvoří virtuální topologii mřížky
  - current_comm – stávající komunikátor
  - dim_num – počet dimenzí
  - dim_sizes – velikost dimenzí
  - wrap – „wrap around“
  - reorder – umožní MPI přiřadit procesy na jednotlivé procesory dle svého nejlepšího vědomí a svědomí

- MPI_COMM_CREATE (IN comm_in, IN group, OUT comm_out)
  - comm_in – stávající komunikátor
  - group – skupina procesů nového komunikátoru
  - comm_out – nový komunikátor

- pvm_staticgroup/pvm_lvgroup

- inter-komunikátory
  - komunikace mezi dvěma procesy – point to point
  - bridge across comm. universes
  - propojení dvou intra-komunikátorů

  - MPI_INTERCOMM_CREATE(
    MPI_Comm local_comm, int local_leader,
    MPI_Comm peer_comm, int remote_leader,
    int tag, MPI_Comm *comm_out)

    - MPI_COMM_SPLIT, MPI_COMM_JOIN
    - MPI_COMM_FREE

- kontext
    - prostředek izolace privátní komunikace skupiny procesů
    - lze zaslat a přijmout zprávu pouze v rámci stejného kontextu
    - rank – id procesu v kontextu
    - MPI_COMM_DUP (pvm_newcontext; od PVM 3.4) – vytvoří nový kontext

Komunikační režim

- Standardní
    - MPI se rozhodne, zda se zpráva nejprve celá zkopíruje do vyrovnávacího bufferu, nebo zda se příjemce s odesílajícím budou synchronizovat
    - vyžaduje kopírování do paměti
    - operace však skončí, jakmile se poslední byte zprávy zkopíruje do bufferu/odešle
    - lze použít send, aniž by příjemce musel zavolat receive
        - ovšem, pokud je zpráva na buffer příliš velká, odesílatel si musí počkat => možnost deadlocku, pokud příjemce neprovede recieve
    - oficiální stanovisko vývojářů MPI: programátor přenositelné aplikace se nezabývá komunikačním režimem a proto používá standardní režim

- buffered communication mode
  - zpráva je vždy kopírována do bufferu, aby odesílající určitě nebyl blokován čekáním na receive
  - pokud se zpráva do bufferu nevejde, funkce vrací chybu

- synchronní
  - send je vždy blokující, dokud zpráva není přijata – tj. příjemce zavolal receive

- ready
  - operace send může proběhnout úspěšně jenom tehdy, když příjemce už předem provedl receive a čeká na zprávu
  - eliminuje potřebu hand-shake, protože příjemce už čeká =>
    - malý výkonnostní nárůst
    - přístup zralý na použití vláken
      - pokud ovšem nemáme programátora, který rád nechává své procesy čekat na data
    - Opravdu to stojí za přehlednost a čitelnost kódu? Nevyplatí se výkonnostní nárůst pokrýt koupí dalšího počítače, když se sečtou hodiny práce nad laděním příslušného kódu, které by se jinak využily lépe?

- funkce používající jiný, než standardní režim, jsou odlišeny prefixem
  - MPI_SEND
  - MPI_BSEND
  - MPI_SSEND
  - MPI_RSEND

Datové typy

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- nejsou popsány pomocí adresy a délky, ale pomocí adresy, délky a id datového typu
- je možné si nadefinovat vlastní type
  - cílem je snížit režii spojenou s používáním velkého množství malých zpráv
    - možnost vícenásobného volán pvm_pack*

- MPI_Datatype – je rekurzivně definován pomocí
  - Předdefinovaných typů – např. MPI_INT
  - Souvislého/krokovaného(strided) pole MPI datatypů
  - Indexovaného pole bloků datatypů
  - Vlastní struktury
    - int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
      - datatype –datový typ
      - extent – velikost datového typu (sizeof)
    - int MPI_Type_struct(int count, int blocklens[], MPI_Aint indices[], MPI_Datatype old_types[], MPI_Datatype *newtype)
      - vytvoří nový datový typ
      - count – velikost polí v jejich prvcích
      - blocklens – počet prvků v každém bloku
      - indices – offsety bloků
      - old_types – datové typy bloků
      - newtype – id nového datového typu
    - MPI_Type_commit

```
/* Some declarations */
typedef struct {
  float  f1,f2,f3,f4;
  int    i1,i2;
  }      f4i2;
f4i2     rbuff, sbuff;
MPI_Datatype newtype, oldtypes[2];
int          blockcounts[2];
MPI_Aint     offsets[2], extent;
MPI_Status   stat;
...

// Setup MPI structured type for the 4 floats and 2 ints
offsets[0] = 0;
oldtypes[0] = MPI_FLOAT;
blockcounts[0] = 4;
MPI_Type_extent(MPI_FLOAT, &extent);
offsets[1] = 4 * extent;
oldtypes[1] = MPI_INT;
blockcounts[1] = 2;
MPI_Type_struct(2, blockcounts,
                offsets, oldtypes, &newtype);
MPI_Type_commit(&newtype);
...

/* Send/Receive 4 floats and 2 ints as a single element*/
for (i=1; i<=REPS; i++){
  MPI_Send(&sbuff, 1, newtype, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(&rbuff, 1, newtype, 1, tag, MPI_COMM_WORLD,
           &stat);
 }
...

/* Send/Receive 4 floats and then 2 ints individually */
for (i=1; i<=REPS; i++){
  MPI_Send(&sbuff.f1, 4, MPI_FLOAT, 1,
                         tag, MPI_COMM_WORLD);
  MPI_Send(&sbuff.i1, 2, MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(&rbuff.f1, 4, MPI_FLOAT, 1,
                         tag, MPI_COMM_WORLD, &stat);
  MPI_Recv(&rbuff.i1, 2, MPI_INT, 1,
                         tag, MPI_COMM_WORLD, &stat);
  }
```

http://www.llnl.gov/computing/tutorials/mpi_performance/

Komunikace mezi dvěma procesy

- int MPI_Send(void *buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm)
  - buf – adresa prvního prvku k odeslání
  - count – počet prvků k odeslání
  - datatype – datový typ prvků k odeslání
  - dest – cílový proces
  - tag – značka zprávy (message tag)
  - comm – komunikátor

- int MPI_Recv(void *buf, int count,
  MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm,
  MPI_Status *status)
  - buf – adresa prvního prvku, kam zkopírovat hodnotu ze zprávy
  - count – počet prvků
    - lze zjistit pomocí MPI_Get_Count
  - datatype – datový typ prvků
  - source – od koho chceme data přijmout
    - lze zadat MPI_ANY_SOURCE
    - rank odesílajícího procesu
  - tag – message tag
    - lze zadat MPI_ANY_TAG
  - comm – komunikátor
  - status – popis výsledku operace
    - OK, pokud se přijme méně než count prvků
    - Přijetí více než count prvků je chyba
      - Buffer overflow, memory corruption?

- int MPI_Sendrecv(void *sendbuf, int sendcount,
  MPI_Datatype sendtype,
  int dest, int sendtag,
  void *recvbuf, int recvcount,
  MPI_Datatype recvtype,
  int source, int recvtag,
  MPI_Comm comm,
  MPI_Status *status)
  - plně duplexní protokol
    - každé odeslané zprávě odpovídá jedna přijatá zpráva

- int MPI_Isend(void *buf, int cnt,
  MPI_Datatype datatype,
  int dest, int tag, MPI_Comm comm,
  MPI_Request *request)
- int MPI_Irecv(void *buf, int cnt, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm,
  MPI_Request *request)
  - asynchronní komunikace
  - operace pouze vytvoří požadavek
    - následně lze zjišťovat stav požadavku
    - anebo jinak pracovat s požadavkem

- MPI_Probe, MPI_IProbe
  - Test na přítomnost nové zprávy

- int MPI_Test(MPI_Request *request, int *flag,
  MPI_Status *status)
- int MPI_Testany(int cnt,
  MPI_Request *array_of_requests,
  int *flag, int *index,
  MPI_Status *status)
- int MPI_Testall(int cnt,
  MPI_Request *array_of_requests,
  int *flag, MPI_Status *array_of_statuses)
  - neblokující testování stavu požadavku
  - flag je true, pokud alespoň jedna (any), nebo všechny (all) operace byly dokončeny

- int MPI_Wait(MPI_Request *request,
  MPI_Status *status)
- int MPI_Waitany(int cnt,
  MPI_Request *array_of_requests,
  int *index, MPI_Status *status)
- int MPI_Waitall(int cnt,
  MPI_Request *array_of_requests,
  MPI_Status *array_of_statuses)
  - blokující čekání na výsledek požadavku

- int MPI_Request_free(MPI_Request *request)
  - uvolnění požadavku

Persistentní komunikace

- obdoba TCP
- snaha eliminovat režii spojenou s vytváření a odesíláním malých zpráv
- obdoba vícenásobného volání pvm_pack*
- konkurence možnosti konstrukce vlastního datového typu
    1. vytvoří se persistentní požadavky
        a. MPI_Send_Init
        b. MPI_Recv_Init

    2. zahájí se přenos
        a. MPI_Start
        b. MPI_StartAll

    3. počká se na dokončení
        a. MPI_Wait*
        b. MPI_Test*

    4. uvolní se asociované prostředky
        a. MPI_Request_Free

```
MPI_Recv_init (&rbuff, n, MPI_CHAR, src, tag,
                                        comm, &reqs[0]);
MPI_Send_init (&sbuff, n, MPI_CHAR, dest, tag,
                                        comm, &reqs[1]);

for (i=1; i <=REPS; i++){
     ...
  MPI_Startall (2, reqs);
     ...
  MPI_Waitall (2, reqs, stats);
     ...
  }
MPI_Request_free (&reqs[0]);
MPI_Request_free (&reqs[1]);
```

http://www.llnl.gov/computing/tutorials/mpi_performance/

Skupinová komunikace (Collective Communications)

- provádějí se současně nad všemi procesy skupiny
- slouží k nahrazení sekvencí send a receive
  - jednak pro pohodlí programátora
  - a jednak z výkonnostních důvodů
- MPI nepředepisuje, jak mají být implementovány
  - výkonnost je tak věcí vývojářů a hardware
    - pro kritické aplikace se vyplatí porovnat, zda to její vývojáři nejsou schopni naprogramovat lépe

- bariéra – int MPI_Barrier (MPI_Comm comm)
- broadcast
  - int MPI_Bcast (void *buffer, int count,
    MPI_Datatype datatype, int root,
    MPI_Comm comm)
  - odešle zprávu od procesu root všem ostatním ve skupině

- all-to-all
  - int MPI_Alltoall(void *sendbuf, int sendcount,
    MPI_Datatype sendtype,
    void *recvbuf, int recvcnt,
    MPI_Datatype recvtype,
    MPI_Comm comm)
  - každý každému
    - každý odešle svou zprávu
    - a přijme zprávy ostatních sdružených do jednoho pole
  - někdy se celá operace nazývá complete exchange

- scatter
  - viz přednáška o PVM
  - int MPI_Scatter (void *sendbuf, int sendcnt,
    MPI_Datatype sendtype,
    void *recvbuf, int recvcnt,
    MPI_Datatype recvtype,
    int root,
    MPI_Comm comm)
  - data přijdou i volajícímu procesu – je v komunikátoru => recvbuffer

- gather
  - viz přednáška o PVM
  - int MPI_Gather (void *sendbuf, int sendcnt,
    MPI_Datatype sendtype,
    void *recvbuf, int recvcount
    MPI_Datatype recvtype,
    int root, MPI_Comm comm)

- reduction
  - viz přednáška o PVM
  - int MPI_Reduce (void *sendbuf, void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op, int root,
    MPI_Comm comm)
    - count – počet prvků v sendbuf
    - recvbuf je jenom jedna hodnota – výsledek
    - Existuje i varianta MPI_AllReduce
      - Výsledek se nepošle jenom rootu, ale všem procesům
- scan – viz SPMD

Souborové operace

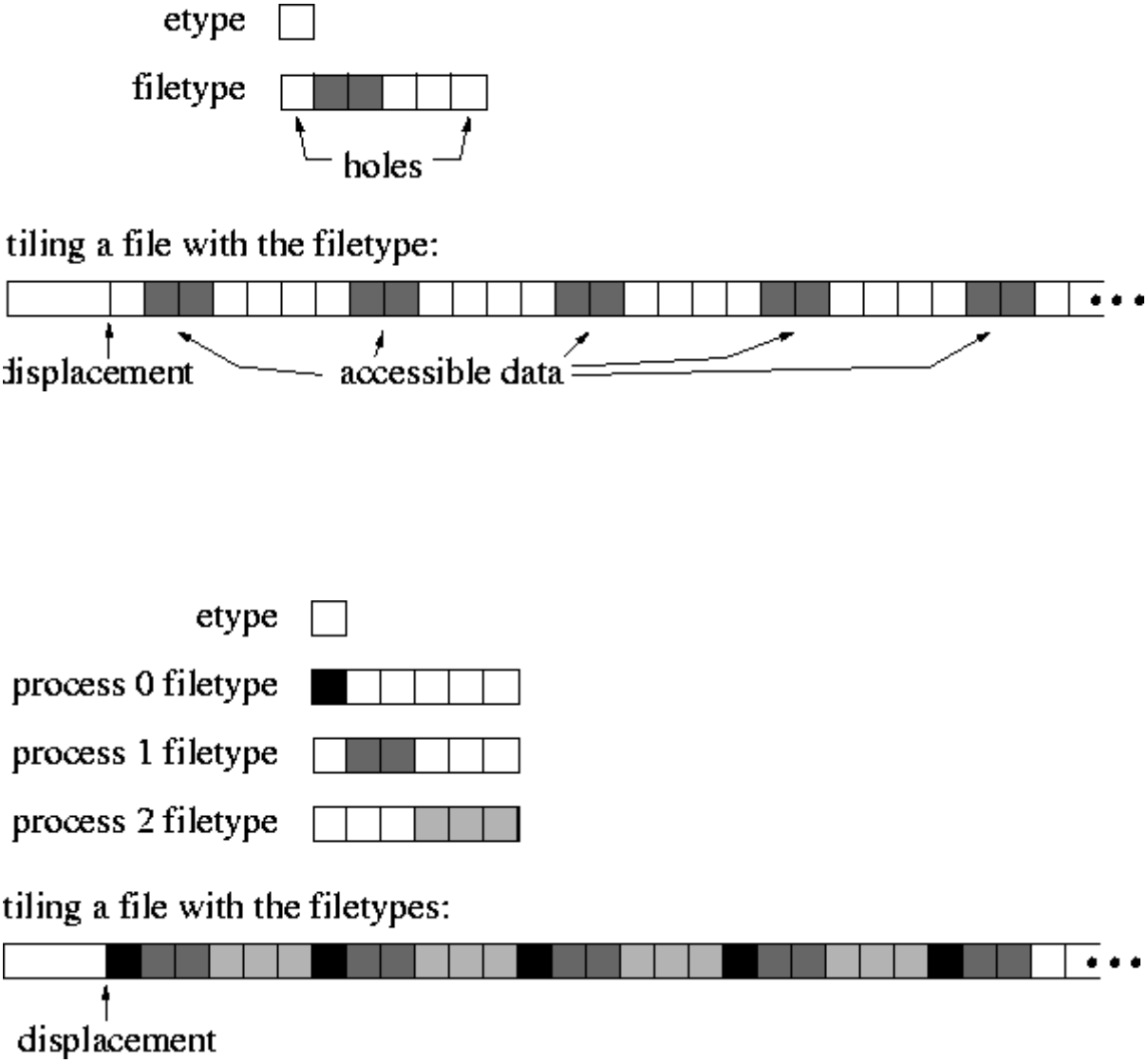
- podpora souborových operací nad jediným souborem prováděných několika procesy současně
- cílem je odstranit úzká místa při datovém toku, která vznikají díky optimalizaci pro ne-paralelní přístup k datům
  - OS přece jenom předpokládá jiný typ procesů, než ty od paralelní aplikace
- soubor se rozdělí na jeden až n bloků
  - bloky nemusí jít lineárně po sobě
- čtení a zápis se může realizovat jako zasílání zpráv

- file
  - MPI soubor je kolekce typovaných datových prvků
  - MPI podporuje náhodný a sekvenční přístup
  - Jeden soubor je obvykle otevřen několika procesy

- displacement – offset od začátku souboru
- etype
  - základní datový typ souboru
  - měří se v něm velikost a pozice v souboru

- filetype – základní datový typ pro „porcování“ souboru mezi procesy, může obsahovat „mezery“, kde nejsou data
- view – množina dat, které proces vidí a může k nim přistupovat
- offset – offset relativní k view, mezery se nepočítají

- file pointer – pozice v souboru, může být sdílená procesy
- file handle – handle, pod kterým se provádějí operace



http://www.mpi-forum.org/docs/mpi-20-html/node173.htm

- int MPI_File_read(MPI_File fh, void *buf, int count,
  MPI_Datatype datatype,
  MPI_Status *status)
- int MPI_File_write (MPI_File fh, void *buffer, int count,
  MPI_Datatype datatype,
  MPI_Status *status);

```
#include <stdio.h>
#include <mpi.h>

typedef struct bitmap {
  int x;
  int y;
  unsigned char* pixels;
} BITMAP;

int mpi_self;  //process number
int mpi_npes;  //number of process
BITMAP picture;//loaded picture
int count;     //size of one part of work


int read_bitmap(char* filename) {
  FILE* file;
  int readcount = 0;
  int size = 0;
  int i = 0;

  if ((file = fopen(filename, "rb"))
      == NULL) {
    printf("File %s not found!\n",
           filename);
    return 0;
  }

  fscanf(file, "%d %d",
              &picture.x, &picture.y);
  size = picture.x * picture.y;
```

```
  if (size == 0) {
    printf("File %s is not correct!\n",
                                filename);
    fclose(file);
    return 0;
  }

  if ((size % mpi_npes) != 0) {
    count = ((size / mpi_npes) + 1)
                                * mpi_npes;
  }else { count = size; }

  picture.pixels = (unsigned char*)
                            malloc(count);

  readcount = fread(picture.pixels, 1,
                     size, file);
  if (readcount != size) {
    printf("File %s is not correct!\n",
                                 filename);
    free((void*) picture.pixels);
    fclose(file);
    return 0;
}

  for (i = 0; i < count - size; i++) {
    picture.pixels[size+i] =
                      picture.pixels[0];
  }
  count = count / mpi_npes;

  fclose(file);
  return 1;
}
```

```
void write_bitmap(char* filename) {
  FILE* file;
  int size = 0;
  int i = 0;

  if ((file = fopen(filename, "wb"))
                                  == NULL) {
    printf("File %s can not be
                  created!\n", filename);
  } else {
    fprintf(file, "%d %d",
                   picture.x, picture.y);
    size = picture.x * picture.y;
    fwrite(picture.pixels, 1,
                              size, file);
    free((void*) picture.pixels);
    printf("Successfully done.\n");
    fclose(file);
  }
}


void stretch_color(void) {
  int i;
  unsigned char min = 0;
  unsigned char max = 0;
  unsigned char lmin = 255;
  unsigned char lmax = 0;
  double konst = 0;
  unsigned char* work = NULL;
  //everyone have to know how much to do
  MPI_Bcast(&count, 1, MPI_INT,
                       0, MPI_COMM_WORLD);
  work = (unsigned char*) malloc(count);
```

```
/* send part of work */
MPI_Scatter(picture.pixels, count,
            MPI_UNSIGNED_CHAR, work,
            count, MPI_UNSIGNED_CHAR,
            0, MPI_COMM_WORLD);

/* compute local extrems */
for (i = 0; i < count; i++) {
  if (work[i] < lmin) {
    lmin = work[i];
  }
  if (work[i] > lmax) {
    lmax = work[i];
  }
}
/* find out global extrems */
MPI_Allreduce(&lmin, &min, 1,
            MPI_UNSIGNED_CHAR, MPI_MIN,
            MPI_COMM_WORLD);
MPI_Allreduce(&lmax, &max, 1,
            MPI_UNSIGNED_CHAR, MPI_MAX,
            MPI_COMM_WORLD);

/* stretch colors */
konst = (255.0 / (max - min));
for (i = 0; i < count; i++) {
  work[i] = (unsigned char)
     (((work[i] - min) * konst) + 0.5);
}
```

```
  /* get whole picture */
  MPI_Gather(work, count,
             MPI_UNSIGNED_CHAR,
             picture.pixels,
             count, MPI_UNSIGNED_CHAR, 0,
             MPI_COMM_WORLD);
  free((void*) work);
}


int main (int argc, char *argv[]) {
  int ok = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,
                &mpi_self);
  MPI_Comm_size(MPI_COMM_WORLD,
                &mpi_npes);

  if (mpi_self == 0) {
    if (argc == 3) {
      ok = read_bitmap(argv[1]);
    } else {
      printf("Incorrect number of
              arguments!\n");
      printf("Arguments are source file
              and destination file.\n");
    }
  }
```

```
  /* waiting for root to read picture */
  MPI_Bcast(&ok, 1, MPI_INT, 0,
            MPI_COMM_WORLD);

  if (ok) {
    stretch_color();
    if (mpi_self == 0) {
      write_bitmap(argv[2]);
    }
  }

  MPI_Finalize();
  return 0;
}
```