Teljesítménynövelés párhuzamosítással

Pörgessük fel és alakítsuk át unalmas soros eljárásainkat úgy, hogy többprocesszoros és megosztott memóriakörnyezetben is futhassanak.

Ebben az írásban azt szemléltetjük, miként lehet átalakítani egy soros eljárást úgy, hogy nagyobb hatékonysággal futhasson szimmetrikus feldolgozást végző (symmetric multiprocessing), illetve megosztott memóriát használó (distributed-memory) környezetekben. Ahhoz, hogy ezt a feladatot teljesíthessük, egyszerű alkalmazást fejlesztünk három lépésben; soros, többszálú és megosztott többszálú változatban.
A párhuzamos programozás elméleti szempontjain túl néhány gyakorlati kérdést is megvizsgálunk, amelyek programozás közben merülhetnek fel. A példákat a C++ és a POSIX-szálak (pthreads) segítségével készítettük el, a szimmetrikus és a megosztott feldolgozáshoz pedig az MPI programkönyvtárat használtuk.

A feladat megfogalmazása

A megoldandó feladat: prímszámok keresése adott határok között. A feladat előnye, hogy egyszerű, ugyanakkor ezen keresztül bemutathatók a párhuzamos programozás fogalmai.

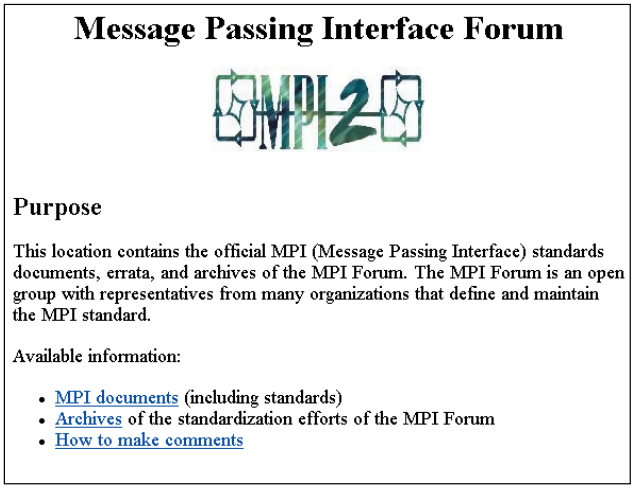
Soros megvalósítás

Az első változatnál úgy döntöttünk, hogy a számok tartományát egy olyan objektummal ábrázoljuk, amely képes az adott tartományban megszámolni a prímszámokat. (lásd az *1. listát*).
A program fordítása és futtatása a következőképpen történhet:

```
bash$ g++ -o primes primes.cpp
bash$ ./primes 0 10000
There were 1229 primes.
```

Többszálú megvalósítás

Ahhoz, hogy a többprocesszoros gépeken növelhessük a végrehajtási sebességet, szálakat kell használnunk. Szeretnénk ragaszkodni az előző változat szerkezetéhez, így meg kell találnunk a módját, hogy minden objektumnak saját végrehajtási szála legyen. A C++ és a pthreads keverése sajnos nem túl könnyű, mivel a `pthread_create( )` C és nem C++ szerkesztésű függvényt vár. Ezt úgy oldottuk meg, hogy készítettünk egy kiegészítő osztályt, illetve egy statikus tagfüggvényt (lásd a *2. listát*).
A `CountPrimes` objektum egyéb részei nem változtak. Három dolgot szükséges itt megjegyezni: 1. A Threaded osztály elvont osztály (abstract class). 2. Az `entry( )` statikus tagfüggvény, ami azt jelenti, hogy ismeri a Threaded osztályt, de nem kötődik annak egy adott példányához. Ezért aztán nem is megy keresztül a névkiértékelésen, így felhasználható C stílusú függvényként. Ez lesz az a függvénymutató, amit átadunk a `pthread_create( )`-nek a „szálasítandó” objektum mutatójával egyetemben. 3. A `run( )` tagfüggvény tisztán virtuális függvény, ezért minden, a Threaded osztályból származtatott osztályban készíteni kell belőle egy változatot. Ez a függvény lesz a származtatott osztály fő végrehajtási pontja, és visszatérési értéke fogja a számítás eredményét megadni. A CountPrimes osztály esetében a függvény egyszerűen kiszámolja és visszaadja a teljes tartományt.
A soros megoldásból szeretnénk minél többet megtartani, ezért csak egyetlen további paramétert (értéket) adunk meg, amely a feladat végrehajtásához felhasználható szálak számát határozza meg. Mivel előre nem tudhatjuk, hány objektumra (szálra) lesz szükségünk, dinamikusan foglalunk le számukra memóriát (lásd a *3. listát*).
E változatban található még néhány ravasz trükk, ezért a kódon egy kicsit részletesebben is végigmegyünk. Először kettőre állítjuk a szálak számának alapértelmezett értékét, majd megvizsgáljuk, hogy a felhasználó nem állított-e be valamilyen más értéket. Készítünk egy pthread_t nevű dinamikus tömböt, amely a szálak azonosítóit fogja tárolni, majd egy másik dinamikus tömböt a CountPrimes objektummutatók számára is. Ez nagyon fontos, mivel valamennyit különböző tartományban kell majd életre hívnunk („példányosítanunk”). Igazából a CountPrimes objektumokból nem is hozhatnánk létre statikus tömböt, hiszen nem határoztunk meg alapértelmezett létrehozó függvényt (konstruktort). Ez a felépítés rákényszerít bennünket, hogy helyesen használjuk az objektumot.



➲ http://www.mpi-forum.org/

Ezután egy ciklus következik, amely az egyes szálakat indítja, hogy ellenőrizzék a megfelelő számtartományokat. Figyeljük meg, hogy eddig még egyáltalán nem foglalkoztunk a terheléselosztással (load balancing). Erre a kérdésre még később visszatérünk. A legfontosabb mozdulat itt az, hogy minden CountPrimes objektum dinamikusan jön létre, mutatóik pedig a fent létrehozott tömbben tárolódnak.
A szálakat tulajdonképpen a `thread_create( )` függvény indítja el, melynek értékként a belépő tagfüggvény mutatóját, illetve az objektumra hivatkozó mutatót adjuk át. A létrehozott szál azonosítója a szálazonosító tömbben tárolódik.
Ezután a `pthread_join( )` és a szálazonosítók segítségével megvárjuk, amíg a szálak végeznek számítási feladatukkal. Mivel `run( )` függvény visszatérési értékének dinamikusan foglaltuk le a helyet, most fel kell szabadítanunk azt. Minden szál visszaadott értékét hozzáadjuk a számlálóhoz.
Végül megsemmisítjük a CountPrimes objektumokat, majd a szálazonosító tömböt és az objektummutató tömböt is. A programot a következőképpen lehet lefordítani és használni:

1. lista Prímek számítása

```
class CountPrimes {
public:
  CountPrimes(long start_, long stop_);
  long total();
private:
  long start;
  long stop;
  long count;
  bool counted;
  bool is_prime (long candidate);
};

CountPrimes::CountPrimes(long start_, long stop_)
  :start(start_),stop(stop_),count(0),
      ➥counted(false){
  if (start>=stop)
    throw range_error("Start >= Stop");
}

bool CountPrimes::is_prime (long candidate){
  // különleges esetben
  if (candidate<0) // negatív
    return false;
  if (!candidate) // == 0?
    return false;
  if (candidate==1) // az 1 nem számít prímnek
    return false;
  if (candidate==2)
    return true;
  // általános esetben
  for (long i=2; i<sqrt(candidate)+1; i++)
    // a jelölt maradék nélkül osztható i-vel?
    if (!(candidate%i))
      return false;
  // ha eddig eljutottunk, a szám prímszám
  return true;
}

long CountPrimes::total(){
  if (counted) // csak egyszer kell számolnunk
    return count;
  for (long i=start; i<=stop; i++)
    if (is_prime(i))
      count++;
  // most, hogy kiszámoltuk, állítsuk a jelzőt
  // igazra
  counted=true;
  return count;
}

// Ezután az objektumot magától értetődő módon
// használhatjuk a parancssorból megkapott
// értékekkel:

int main (int argc, char *argv[]){
  if (argc<3)
    usage(argv[0]);

  try {
    CountPrimes
      ➥counter(atol(argv[1]),atol(argv[2]));
    if (counter.total()>1){
      cout << "There were " << counter.total();
      cout << " primes." << endl;
    }
    else{
      cout << "There was " << counter.total();
      cout << " prime." << endl;
}
```

```
bash$ g++ -o primes_threaded
➥primes_threaded.cpp-lpthread
bash$ ./primes_threaded 0 10000 4
There were 1229 primes.
```

Megosztott megvalósítás

Az üzenetátadó felület (Message passing interface avagy MPI) a megosztott programok általános programozási felülete (API). Az MPI használatának sok előnye van, de a legnagyobb mégis az, hogy forrásszinten mindig megfelelő marad, függetlenül attól, hogy éppen milyen MPI-megvalósítást használunk. A példában mi a Notre Dame MPI-változatára (LAM – local area multicomputer, lásd a *Kapcsolódó címeket*) hivatkozunk és feltételezzük, hogy azt helyesen állították be.

A megosztott programozás egyik általános módja a mester-szolga modellre épül. Ebben a modellben az egyik folyamatot mesternek nevezik. Ez a folyamat osztja szét a munkát a szolgák között. A szolgák üzennek a mesternek, ha elkészültek, és új feladatot kérnek, amennyiben még van elvégzendő munka. Ez az egyszerű elvű modell nagyon jól működik, ha a feladat nem igényel komolyabb összehangolást és a szolgák teljesen függetlenek lehetnek. Ezt a feladattípust gyakran „kényszerpárhuzamosítottnak” is nevezik (embarrassingly parallel).

Az új program elkészítéséhez először el kell döntenünk, hogyan alakítsuk át a korábbi változatot a mester-szolga modellnek megfelelően, és be kell iktatnunk a szükséges MPI-hívásokat, hogy megosszuk a feladatot és begyűjtsük az eredményt. A *4. lista* bemutatja, milyen változtatások szükségesek a `main()` függvényben.

A megosztott program elején meg kell hívnunk az `MPI_Init( )` függvényt, hogy felvehessük a kapcsolatot a többszámítógépes rendszerrel (multicomputer). A következő két függvény a rangunkat és a számításban részt vevő számítógépek számát állapítja meg. Az MPI a rendszer valamennyi számítógépén ugyanazt a programot indítja el. Ezért szükséges futásidőben meghatároznunk a rangunkat, hogy eldönthessük, mesterek vagy szolgák vagyunk-e. A `master( )` vagy a `slave( )` függvényt rangunktól függően hívjuk meg. Miután végeztünk a számítással, meg kell hívnunk az `MPI_Finalize( )`-t, hogy elszakadjunk a rendszertől. A `slave( )` függvény mindössze egyetlen értéket vár, nevezetesen a felhasználandó szálak számát. Így a telepben található többprocesszoros (SMP) gépek teljes számítási kapacitását kihasználhatjuk.

A szolgák feladata, hogy munkára várakozva üldögéljenek, végrehajtsák a munkát és visszaadják az eredményt. Ezt kell folytatniuk mindaddig, amíg meg nem kapják a jelzést, hogy nincs több munka (lásd az *5. listát*).

A `slave( )` függvény kódjának nagy része hasonló az előző

2. lista Kiegészítő osztály és statikus tagfüggvény készítése

```
class Threaded {
public:
  Threaded() {};
  virtual ~Threaded() {};
  static void *entry (void *ptr);
  virtual void *run ()=0;
};

void *Threaded::entry(void *ptr){
  return reinterpret_cast<Threaded*>
       ➥(ptr)->run();
}

class CountPrimes : public Threaded {
public:
  CountPrimes(long start_, long stop_);
  long total();
  void *run();
private:
  long start;
  long stop;
  long count;
  bool counted;
  bool is_prime (long candidate);
};

void *CountPrimes::run(){
  long *return_val=new long(total());
  return reinterpret_cast<void*>(return_val);
}
```

„szálasított” változat `main( )` függvényéhez. Az egyetlen különbség az, ahogyan a szolga a számlálandó prímek határértékeit megkapja, illetve ahogyan az eredményt visszaadja.

A szolgák végtelen ciklusban várakoznak a mestertől érkező munkára, amit a `MPI_Recv()` függvényen keresztül kapnak meg. Ez a függvény a mester által küldött két hosszú egész számot (long integer, long) várja, melyek a feldolgozandó tartomány határértékeit adják meg. Miután a mestertől átvette az adatot, a szolga ellenőrzi az üzenet állapotát, hogy lássa, a munka véget ért-e már (KILL üzenet), amennyiben igen, befejezi a működését. Ha még van munka, a változókat átnevezzük, hogy pontosan az előző változatban lévő kódot használhassuk. Az egyetlen hátralévő lépés, hogy az eredményt visszaküldjük az `MPI_Send( )`-en keresztül. Itt egyetlen long értéket küldünk vissza, amelyben a szolga által talált számok darabszámát tároljuk.

A mester feladata valamivel összetettebb, mivel el kell döntenie, hogyan ossza fel a szolgáknak kiküldendő munkát és hogyan gyűjtse be az eredményt. Először kiküldi a munka egy részét a szolgáknak, majd megvárja, amíg az eredmény visszaér hozzá. Amikor a mester megkapja az eredményt, egy másik munkaegységet küld ugyanannak a folyamatnak, feltéve, hogy van még hátralévő munka. Ha nincs több, minden folyamatot még egyszer végigkérdez, nem maradt-e esetleg valamilyen eredmény, majd minden szolgának kiadja a kilépési utasítást (lásd a *6. listát*).

A `make_work( )` függvény dönti el, hogyan kell felosztani a munkát és mikor van annak vége. Egyszerű soros modellt választottunk, ahol a darabok méretét a `STEP_SIZE` határozza meg. (lásd a *7. listát*).

3. lista Szálak létrehozása

```
int main (int argc, char *argv[]){
  int num_threads(2);
  // helyesen hívtak meg?
  if (argc<3)
    usage(argv[0]);
  if (argc==4)
    num_threads=atol(argv[3]);

  try {
    pthread_t *t_id=new pthread_t[num_threads];
    CountPrimes **counter=
                 new CountPrimes*[num_threads];
    // indítsuk a szálakat
    long start(atol(argv[1]));
    long stop(atol(argv[2]));
    long incr((stop-start)/num_threads);
    for (int i=0; i<num_threads; i++){
      counter[i]=new CountPrimes(start,
        ➥(start+incr)>stop?stop:start+incr);
      start+=incr+1;
      pthread_create(&t_id[i],NULL,
                 counter[i]->entry,counter[i]);
    }
    // most várjuk meg az eredményt
    long count=0;
    for (int i=0; i<num_threads; i++){
      void *return_val;
      pthread_join(t_id[i],&return_val);

count+=*(reinterpret_cast<long*>(return_val));
      delete reinterpret_cast<long*>(return_val);
    }
    for (int i=0; i<num_threads; i++)
      delete counter[i];

    delete[] counter;
    delete[] t_id;

    if (count>1){
      cout << "There were " << count;
      cout << " primes." << endl;
    }
    else{
      cout << "There was " << count;
      cout << " prime." << endl;
    }
  }
  catch (range_error e){
    cout << "Exception: " << e.what() << endl;
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

Az egyes gépek közötti terheléskiegyenlítés kulcsa a `STEP_SIZE` változó. Ha az értéke túl nagy, egyes gépek üresjáratba kerülhetnek, míg mások túlságosan nagy tartományokkal bajlódnak. Ha viszont túl kicsi, akkor túl nagy lesz a kapcsolati terhelés. Az értéket általában a legegyszerűbb tapasztalati úton meghatározni. A részletekkel

5. lista Szolgák

```
void slave(int num_threads){
  long result;
  long bounds[2];
  MPI_Status status;

  while(true){
    MPI_Recv(bounds, 2, MPI_LONG, 0,
      ➥MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    if (status.MPI_TAG == KILL)
      return;

    try {
      long start(bounds[0]);
      long stop(bounds[1]);

/*... ide jön a szálasított változat main()
      függvényének kódja ...*/

      MPI_Send(&count,1,MPI_LONG, 0,
                    0, MPI_COMM_WORLD);
    }
    catch (range_error e){
      cout << "Exception: " << e.what() << endl;
   }
  }
}
```

4. lista A main() változásai

```
int main (int argc, char *argv[]){
  int num_threads(2);
  int my_rank;
  int nprocs;

  // MPI előkészítése
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  // helyesen hívtak meg?
  if (argc<3)
    usage(argv[0]);
  if (argc==4)
    num_threads=atol(argv[3]);
  long start(atol(argv[1]));
  long stop(atol(argv[2]));

  if (my_rank == 0)
    master(start,stop,nprocs);
  else
    slave(num_threads);

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```


a Teljesítmény részben bővebben is foglalkozunk.
Az MPI programok fordításához az mpicc vagy az mpiCC használható, attól függően, hogy C vagy C++ kódot fordítunk-e. A megosztott program futtatásához először be kell indítani a többgépes rendszert a `lamboot`-on keresztül, majd a programot elindítani az `mpirun` paranccsal. Az MPI-munkamenet végén a rendszert a `wipe` paranccsal állíthatjuk le:

```
bash$ mpiCC -O -o primes_mpi
primes_mpi.cpp -lpthread
bash$ lamboot

LAM 6.3.2/MPI 2 C++/ROMIO
-University of Notre Dame

bash$ mpirun -O -np 16 primes_mpi
➥-- 0 10000000
There were 664579 primes.
bash$ wipe
```

Ha a `lamboot` futtatásával gondok akadnak, használjuk a `recon` parancsot annak kiderítésére, mi is okozza a hibát. Ha a `recon` sikertelen, valószínűleg nem futtathatunk programokat a távoli gépeken jelszó begépelése nélkül. Ha ssh-t használunk, bizonyosodjunk meg róla, hogy ennek jelzésére beállítottuk a `LAMRSH`-t:

```
bash$ export LAMRSH='which ssh'
```

Az mpicc kapcsolói lényegében ugyanazok, mint amelyeket egyébként a fordítónak szoktunk átadni. Az egyetlen kivétel mind a mpicc, mind az mpirun esetében a `-O`, amely azt mutatja, hogy a rendszer azonos típusú számítógépekből áll, így endian átalakításra nincs szükség.

A mpirun `-np` kapcsolója az elindítandó feladatok számát határozza meg (azaz általában a rendszer csomópontjainak számát). A két mínuszjel (`--`) utáni minden további érték átadódik a futó főprogramnak.

Teljesítmény

A párhuzamos programozás hatékonyságának bemutatásához meg kell mutatnunk, hogy az eltelt idő a párhuzamos változatban rövidebb. A csomópontonkénti százszázalékos teljesítménynövekedés általában nem érhető el, kivéve ha a feladatot a rendszer nagy darabokra bontja fel és komolyabb összehangolásra sincs szükség. Próbáinkat 16 két 700 MHz-es Pentium III-as processzort és 384 MB memóriát tartalmazó gépből álló telepen futtattuk. A programmal 0 és 10 000 000 között számítottuk ki a prímeket. Íme az egyes programok futási eredményei:

- Soros megvalósítás egyetlen csomóponton: 6:29,28 mp.
- Többszálú megvalósítás egyetlen csomóponton: 3:24,24 mp.
- Megosztott (és többszálú) megvalósítás 16 csomóponton: 11,05 másodperc.

Ezek az eredmények azt mutatják, hogy a processzorok száma egyenesen arányos a sebességgel (32x-es sebességgyorsulás a soros megvalósításhoz képest).

Terheléselosztás

A többgépes rendszerek programozásakor az egyik legnagyobb gond, hogy miként használhatjuk ki a lehető legjobban az összes számítógépet, illetve a többprocesszoros gépek összes processzorát. Nyilván el szeretnénk kerülni, hogy néhány gép üresjáratban várjon más számítások eredményére, melyeket egy másik gép vagy processzor végez. Ezt a finom művészetet nevezik terheléselosztásnak (load balancing).

6 lista Szolgák kiléptetése

```
void master(long start, long stop, long nprocs){
  long work[2];
  long result;
  long total(0);
  long current(0);
  MPI_Status status;

  // osszunk ki némi munkát
  for (int rank=1; rank<nprocs; ++rank){
    // határok beállítása ehhez a munkához
    if (make_work(work,&current,stop))
      MPI_Send(work,2,MPI_LONG,
               rank,WORK,MPI_COMM_WORLD);
  }
  // küldjünk még munkát, ha van
  while(make_work(work,&current,stop)){
    MPI_Recv(&result,1,MPI_LONG,MPI_ANY_SOURCE,
      ➥MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    total+=result;
    MPI_Send(work,2,MPI_LONG,status.MPI_SOURCE,
      ➥WORK,MPI_COMM_WORLD);
  }
  // fogadjuk a különleges kérelmeket
  for (int rank=1; rank<nprocs; ++rank){
    MPI_Recv(&result,1,MPI_LONG,MPI_ANY_SOURCE,
      ➥MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    total+=result;
  }
  // utasítsuk kilépésre a szolgákat
  for (int rank=1; rank<nprocs; ++rank){

MPI_Send(0,0,MPI_INT,rank,KILL,MPI_COMM_WORLD);
  }
  cout << "There were " << total;
  cout << " primes." << endl;
}
```

A terheléselosztás részletes ismertetése természetesen meghaladja cikkünk kereteit, de a megoldandó feladat néhány részletét azért megvizsgálhatjuk, hogy megtanuljuk, miképpen növelhetjük a teljesítményt. Példánkban a számítások nagy részét az `is_prime( )` függvény végzi. Természeténél fogva végrehajtási ideje nagyban függ a bemenő értékek számától. Figyeljük meg, hogyan osztottuk fel a munkát a második megoldásban, amikor csak két szálat használtunk: a számok felét az egyik, másik felét a másik szálnak adtuk át. Ez eredendően kiegyensúlyozatlan, hiszen a számokat sorban osztottuk el. A kisebb számokat kapó szál sokkal hamarabb fog végezni, mint a nagyobbakon dolgozó, így a processzor üresjáratba kerül. A gond legalább két módon megoldható: amikor felosztjuk a számtartományt, az egyes szálaknak felváltva küldjük a számokat; vagy egyszerűen még több szálat használunk, ami kisebb darabokra osztja fel a feladatot, és nagyobb mértékben bízza a terheléselosztást a rendszermag ütemezőjére. Természetesen ez csak addig működőképes, amíg az ütemezéssel töltött idő meg nem haladja a feladat felosztásával megtakarított időt.

A terhelés elosztása lényegesen hatékonyabb volt a megosztott megoldásban: minden gépnek csak kisebb munkaszakaszokat küldtünk, és csak akkor kaptak újabbat, ha az előzővel már végeztek. A kiküldött (a mi megoldásunkban a `STEP_SIZE` változó által meg-

7. lista. Soros megoldás

```
bool make_work(long *work, long *current,
       ➥long stop){
  if (*current>=stop)
    return false; // nincs több munka

  work[0]=*current;
  work[1]=*current+STEP_SIZE;
  *current=*current+STEP_SIZE+1;

  if (work[1]>stop)
    // ügyeljünk rá, nehogy átlépjük a határt
    work[1]=stop;
  return true;
}
```

határozott) darabok méretére azért oda kell figyelni, mivel jelentősen megnövelhetjük a hálózati forgalmat – tényleges sebességnövekedés nélkül. Hasonló megoldást alkalmazhattunk volna a szálak kiegyensúlyozására is, az átláthatóság kedvéért azonban nem ezt tettük.

Eric Bourque
(ericb@computer.org) PhD hallgató Kanadában, a montréal-i McGill Egyetemen. A számítógépes grafika és a képalapú mintázatrajzolás problémáival foglalkozik. Rendelkezik egy szakirányú (MSc) és egy zenei (szaxofon BMus) diplomával is.

Kapcsolódó címek

Jó angol nyelvű C++ könyvek gyűjteménye
➲ http://www.telegraph-road.org/books.html

Bjarne Stroustrup, a C++ megalkotójának könyve, a C++ programozási nyelv magyar változatának honlapja
➲ http://www.zolix.hu/konyv_hu.html

Linux Paralell Processing HOWTO
➲ http://www.beta.ttt.bme.hu/HOWTO/Paralell-Processing-HOWTO.html
➲ http://www.linuxdoc.org/HOWTO/Paralell-Processing-HOWTO.html

C++ FAQ
➲ http://www.parashift.com/c++-faq-lite/

comp.programming.threads FAQ
➲ http://www.LambdaCS.com/newsgroup/FAQ.html

MPI Forum
➲ http://www.mpi-forum.org/

LAM
➲ http://www.mpi.nd.edu/lam

LAM FAQ
➲ http://www.mpi.nd.edu/lam/faq/

A cikk teljes forráskódja megtalálható a következő címen:
➲ ftp://ftp.cim.mcgill.ca/pub/people/ericb/primes.tar.gz