Parallelizing Modern Codes with OpenMP, MPI, C++11, HPX

Dr. Steven R. Brandt

Center for Computation and Technology
Louisiana State University, Baton Rouge, LA

June 23, 2016

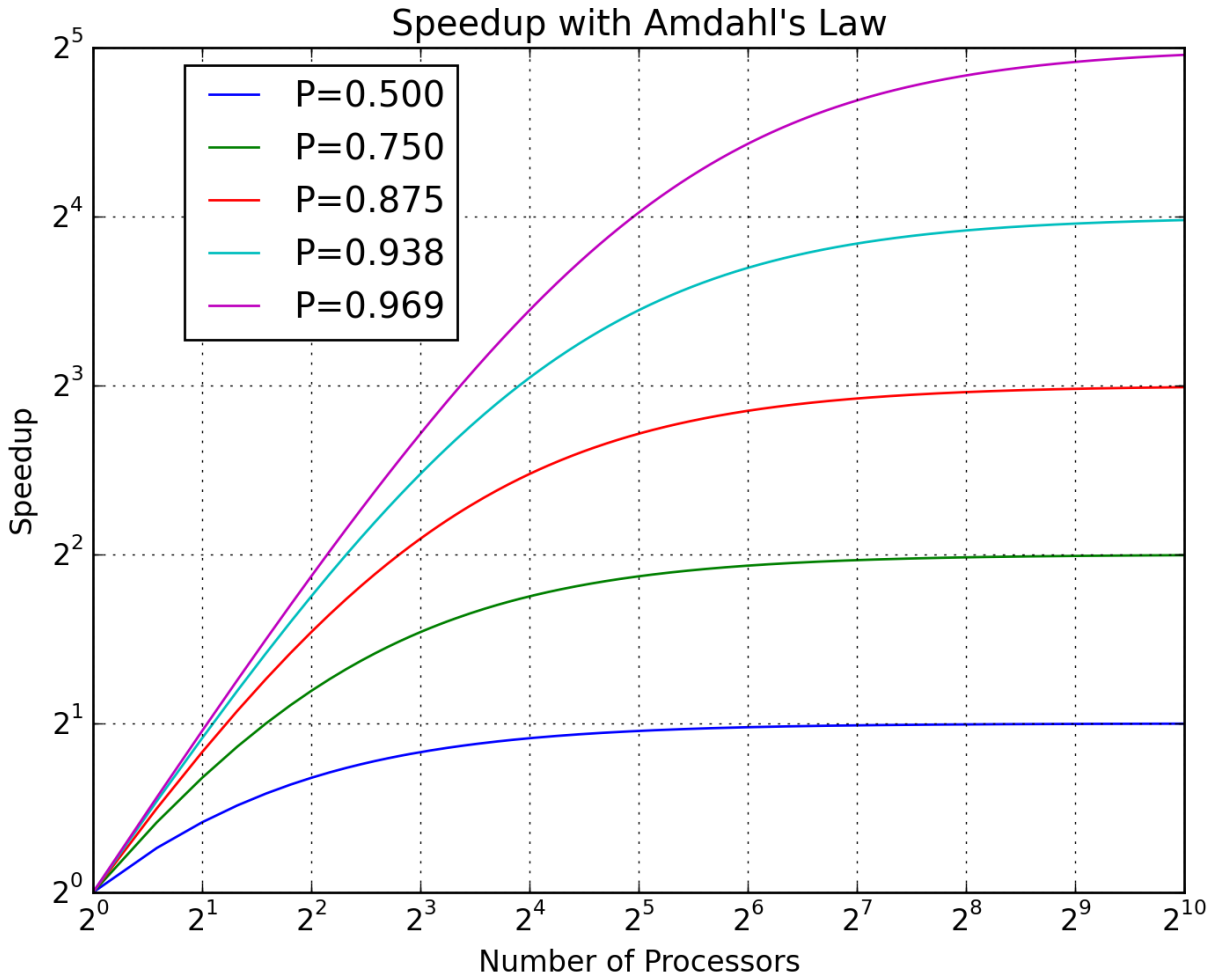
The Problem

Amdahl's Law

Let P be the fraction of time spent in a parallel region
Let T be the number of seconds your code runs without parallelism
Let S be the speedup of your code
Let N be the number of cores you have

$$
\begin{aligned}
T_P &= T(1-P) + TP/N \\
S &= \frac{T}{T_P} \\
S &= \frac{1}{1-P+P/N}
\end{aligned}
$$

Amdahl's Law


Speedup with Amdahl's Law
P=0.500
P=0.750
P=0.875
P=0.938
P=0.969
Speedup
Number of Processors
2^0
2^1
2^2
2^3
2^4
2^5
2^6
2^7
2^8
2^9
2^{10}

Modern Machines

How much parallelism do we need to use?

- LSU
  - QueenBee-II - 10,080 cores: 504 nodes, two 10-core processors per node (23,040 cores with accelerators)
  - SuperMic - 7,200 cores: 360 nodes, two 10-core processors per node (45,866 cores with accelerators)
- Elsewhere
  - Stampede - 102,400 cores: 6,400 nodes, two 8-core processors per node (462,462 cores with accelerators)
  - Tianhe2 - 192,000 cores: 16,000 nodes, 12 cores per node (3,120,000 cores with accelerators)
  - Sunway TaihuLight - 163,840 cores: 40,960 nodes?, 4 cores per node (10,649,600 cores with onchip accelerators)

Incrementally Adding Parallelism

OpenMP allows you to take an existing non-parallel code and add parallelism incrementally by adding pragmas. For example, if you have a loop:

```
for(int k=0;k<N;k++) {
  #pragma omp parallel for
  for(int j=0;j<N;j++) {
    c[k][j] = 0;
    for(int i=0;i<N;i++) {
      c[k][j] += a[k][i]*b[i][j];
    }
  }
}
```

Incrementally Adding Parallelism

- OpenMP only enables you to parallelize within a single node. By itself, it does not make it possible for you to use a large machine.
- If the compiler flags for openmp are not enabled, the pragma directives will be ignored and your program will run sequentially.
- Directives can be used in Fortran as well. In that case you write "!$OMP parallel for" instead of a pragma.
- Caution: Adding parallelism also adds overhead and can make your code run *slower.*
- Caution: If one iteration of a loop depends on another, making it run in parallel will result in incorrect results.
- This presentation will just scratch the surface of what's possible.

Incrementally Adding Parallelism

You can control the amount of parallelism (number of threads). Getting the most out of your code may depend on getting the right *grain size* for your problem.

```
int n = 2; // figure out how many threads
#pragma omp parallel num_threads(n)
#pragma omp for
for(int i=0;i<len;i++)
  do_work(i)
```

Incrementally Adding Parallelism

By default, all variables are considered to be shared between threads. Declare them private if you don't want this.

```
double sum;
for(int k=0;k<N;k++) {
  #pragma omp parallel for private(sum)
  for(int j=0;j<N;j++) {
    sum = 0;
    for(int i=0;i<N;i++) {
      sum += a[k][i]*b[i][j];
    }
    C[k][j] = sum;
  }
}
```

Incrementally Adding Parallelism

Some inter loop dependencies can handled within OpenMP. Most important of these are reductions, i.e. operators +, *, |, &, ^, max, and min. Often, these may be implemented in a particularly efficient manner.

```
   double *d = ...
   double sumd = 0;
   #pragma omp parallel
   #pragma omp for reduction(+:sumd)
   for(int i=0;i<len;i++) {
      sumd += d[i];
   }
}
```

Incrementally Adding Parallelism

The following is a *far* less efficient implementation of the same program.
No two threads can execute the same critical directive at the same time.

```
double *d = ...
double sumd = 0;
#pragma omp parallel
#pragma omp for
for(int i=0;i<len;i++) {
  #pragma omp critical(update_sumd)
  sumd += d[i];
}
```

Incrementally Adding Parallelism

Sometimes you want to run tasks in parallel rather than parts of a loop. You can do that in this way:

```
#pragma omp sections  // parallel tasks
                      // all three below
                      // can run in parallel
{
  { task1(); }
  #pragma omp section
  { task2(); // task 3 must run after 2
    task3(); }
  #pragma omp section
  { task4(); }
}
```

Copied from: http://bisqwit.iki.fi/story/howto/openmp/

Using MPI

- MPI stands for "Message Passing Interface."
- OpenMP offers an easy path to getting some parallelism using pragmas, MPI provides a less easy path using a standard library.
- OpenMP operates in shared memory. MPI operates on distributed memory. That means thinking about placement of memory on remote machines.

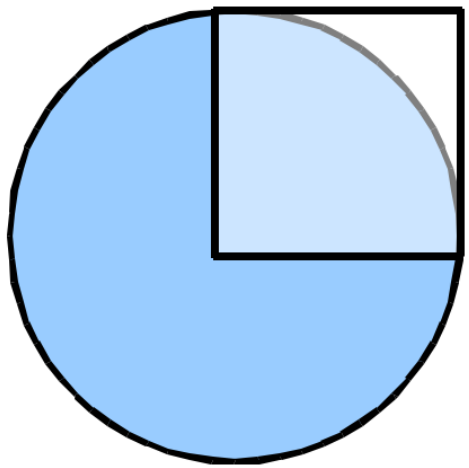
A Minimal MPI program...

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

int main(int argc,char **argv) {
  int rank, size;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  std::cout << "rank=" << rank << " size=" <<
     size << std::endl;
  MPI_Finalize();
  return 0;
}
```

Using MPI

- Generate random points in a box
- Find fraction that fall inside the circle = f
- Inside: $x^2+y^2 < 1$
- Area of a Quarter of a Circle = A = $\pi r^2/4$
- Area of Box = B = r^2
- A/B = f = $\pi/4$
- $\pi = 4 f$



Computing PI with OpenMP

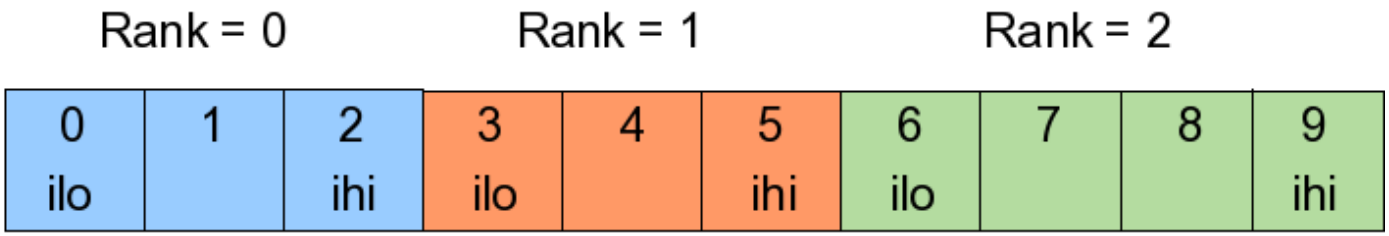
```
int ndarts = 1200000, inside = 0;
#pragma omp parallel for reduction(+:inside)
for(int i=0;i<ndarts;i++) {
  double x = (1.0*rand())/RAND_MAX;
  double y = (1.0*rand())/RAND_MAX;
  if(x*x+y*y <= 1.0) inside++; }
std::cout << "inside=" << inside
 << " pi=" << (4.0*inside)/ndarts << std::endl;
```

Computing PI with MPI

```
int rank, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
int ndarts=1200000/size,inside=0,g_inside;
for(int i=0;i<ndarts;i++) {
  double x = (1.0*rand())/RAND_MAX;
  double y = (1.0*rand())/RAND_MAX;
  if(x*x+y*y <= 1.0) inside++; }
MPI_Allreduce(&inside,&g_inside,1,
  MPI_INT,MPI_SUM,MPI_COMM_WORLD);
if(rank==0)std::cout << "inside=" << g_inside
   << " pi=" << (4.0*g_inside)/ndarts/
   size << std::endl; MPI_Finalize();
```

A Simplistic Way to Divide Work...Evenly

N = num_items/size =
10 items / 3 processes = 3 items per process



ilo = N*Rank
ihi = N*(Rank+1)-1
If Rank + 1 == Size:
ihi = num_items -1

The Heat Equation... Sort Of

```
#include <iostream>
#include <stdlib.h>
int main(int argc,char **argv) {
  const int N = 10;
  double *d = new double[N], *tmp;
  double *d_next = new double[N];
  for(int i=0;i<N;i++)
    d_next[i] = d[i] = (1.0*rand())/RAND_MAX;
  for(int t=0;t<10000;t++) {
    #pragma omp parallel for
    for(int i=1;i<N-1;i++)
      d_next[i] = 0.5*(d[i+1]+d[i-1]);
    tmp=d; d=d_next;d_next=tmp; } }
```

Zoom In on the Loop

```
for(int t=0;t<5;t++) {
  #pragma omp parallel for
  for(int i=1;i<N-1;i++)
    d_next[i] = 0.5*(d[i+1]+d[i-1]);
  tmp=d; d=d_next;d_next=tmp; // swap
}
for(int i=1;i<N-1;i++)
  std::cout << d[i] << " ";
std::cout << std::endl;
```

Convert to MPI

```
if(rank==0) ilo = 1;
if(rank+1==size) ihi = N-2;
for(int t=0;t<10000;t++) {
  if(rank+1 < size) MPI_Isend(d+ihi,1,
    MPI_DOUBLE, rank+1,0,MPI_COMM_WORLD,&req);
  if(rank > 0) MPI_Recv(d+ilo-1,1,
    MPI_DOUBLE, rank-1,0,MPI_COMM_WORLD,&stat);
  if(rank > 0) MPI_Isend(d+ilo,1,
    MPI_DOUBLE, rank-1,0,MPI_COMM_WORLD,&req);
  if(rank+1 < size) MPI_Recv(d+ihi+1,1,
    MPI_DOUBLE, rank+1,0,MPI_COMM_WORLD,&stat);
  MPI_Barrier(MPI_COMM_WORLD);
  for(int i=ilo;i<=ihi;i++)
    d_next[i] = 0.5*(d[i+1]+d[i-1]);
  tmp=d; d=d_next;d_next=tmp; // swap
}
for(int i=0;i<size;i++) {
  if(i==rank) {
    for(int i=ilo;i<=ihi;i++)
      std::cout << d[i] << " ";
    if(rank+1==size) std::cout << std::endl;
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

Some Notes...

- The complex code inside the loop is to exchange boundary data, or ghost zones.
- The previous listing is simplified, it ignores the memory allocation.
- While MPI is more complex, it's needed to make use of big machines. OpenMP can, at best, get you some of the power of a single node.
- Like OpenMP, we've only scratched the surface.

C++11 Makes Asynchronous Programming Possible

In the code below, task1 and task2 run in parallel. When get() is called on fut1, the main thread blocks and waits for it to finish. Futures can be used to hide latency and provide load balancing.

```
#include <future>
int task1() { ... }
int task2() { ... }
int main() {
  std::future<int> fut1 = std::async (
    std::launch::async ,task1);
  int result2 = task2();
  int result1 = fut1.get(); // waits for task to
     finish
}
```

C++11 Makes Asynchronous Programming Possible

This is the same as the code above, except that we use a lambda function to do task1.

```
#include <future>
int main() {
  std::future<int> fut1 = std::async (
    std::launch::async ,[](){ ... /* task 1 */
      ... });
  int result2 = task2();
  int result1 = fut1.get(); // waits for task to
     finish
}
```

C++11 Makes Asynchronous Programming Possible

A Simple Loop

```
void task(int i) { ... }
int main() {
  const int n = 10;
  for(int i=0;i<n;i++) { task(i); }
```

A Parallel Loop

```
#include <future>
void task(int i) { ... }
int main() {
  int nthreads = std::thread::hardware_concurrency();
  const int n = 10;
  std::future<void> futs[nthreads];
  for(int th=0;th<nthreads;th++) {
    futs[th] = std::async(std::launch::async,[n,
        nthreads,&futs,th](){
      for(int i=th;i<n;i += nthreads) task(i); }); }
  for(int th=0;th<nthreads;th++) futs[th].wait(); }
```

A Parallel Loop Again...

Note that you can capture all the logic above into a function named par_for and be done with it.

```
void task(int i) { ... }
int main() {
  int nthreads = std::thread::
     hardware_concurrency();
  // Serial loop
  const int n = 10;
  for(int i=0;i<n;i++) { task(i); }
  // Parallel loop
  par_for(0,n,task);
}
```

C++11 Computing PI

```
#include <iostream>
#include <future>
int main(int argc,char **argv) {
  unsigned nthreads = std::thread::hardware_concurrency();
  int ndarts = 1200000/nthreads;
  int global_inside = 0;
  std::future<int> futs[nthreads];
  for(int i=0;i<nthreads;i++) {
    futs[i] = std::async ( std::launch::async,[ndarts](){
      int inside = 0; // begin lambda function
      for(int i=0;i<ndarts;i++) {
        double x = (1.0*rand())/RAND_MAX;
        double y = (1.0*rand())/RAND_MAX;
        if(x*x+y*y <= 1.0)
          inside++; }
      return inside; // end lambda function
    }); }
  for(int i=0;i<nthreads;i++) {
    global_inside += futs[i].get(); }
  std::cout << "inside=" << global_inside
      << " pi=" << (4.0*global_inside)/ndarts/nthreads
      << std::endl; }
```

Introducing HPX

- Fully compatible with the C++11/14/17 standard
- Makes additional functionality available
- Is a vehicle for moving the standard forward
- Supports distributed programming
- Developed at CCT and internationally
- See http://stellar-group.org/

HPX

You can use futures, just like in C++11, but they're in the hpx namespace.

```
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <hpx/include/iostreams.hpp>

int task() { return 123; }

int main(int argc,char **argv) {
  hpx::future<int> f =
    hpx::async(hpx::launch::async,task);
  hpx::cout << "f=" << f.get() << hpx::endl;
  return 0;
}
```

HPX

You can schedule tasks to run asynchronously when the future is ready using "then."

```
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <hpx/include/iostreams.hpp>

int task() { return 123; }
void later(int n) {
  hpx::cout << "f=" << n << hpx::endl;
}
int main(int argc,char **argv) {
  hpx::future<int> f =
    hpx::async(hpx::launch::async,task);
  f.then(hpx::util::unwrapped(later));
  return 0;
}
```

HPX

You can compose futures with "when_all", "when_any" or "wait_all"

```
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <hpx/include/iostreams.hpp>
int task1() { return 123; }
int task2() { return 234; }
int main(int argc,char **argv) {
  hpx::future<int> f1 =
    hpx::async(hpx::launch::async,task1);
  hpx::future<int> f2 =
    hpx::async(hpx::launch::async,task2);
  hpx::when_all(f1,f2).then(hpx::util::unwrapped([](
      auto arg) {
    hpx::cout << hpx::util::get<0>(arg).get() << ","
              << hpx::util::get<1>(arg).get()
              << hpx::endl; })); return 0; }
```

HPX

This looks a little nicer.

```
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <hpx/include/iostreams.hpp>
int task1() { return 123; }
int task2() { return 234; }
int main(int argc,char **argv) {
  hpx::future<int> f1 =
    hpx::async(hpx::launch::async,task1);
  hpx::future<int> f2 =
    hpx::async(hpx::launch::async,task2);
  hpx::dataflow(hpx::util::unwrapped(
    [](int a,int b) {
      hpx::cout << a << "," << b << hpx::endl;
    }),f1,f2);
  return 0; }
```

HPX

Parallel algorithms

```
...
#include <hpx/include/parallel_for_each.hpp>
#include <hpx/parallel/algorithms/transform.hpp>
#include <boost/iterator/counting_iterator.hpp>
int main(int argc,char **argv) {
  std::vector<int> v(10);
  // Initialize a vector
  auto f = hpx::parallel::transform(
    hpx::parallel::par(hpx::parallel::task),
    boost::counting_iterator<int>(0),
    boost::counting_iterator<int>(v.size()),
    v.begin(),
    [](auto i)->int { return i; });
  f.wait();
  return 0;
}
```

HPX

Parallel algorithms

```
...
hpx::parallel::transform(hpx::parallel::par,
  v.begin(),v.end(),v.begin(),
  [](auto i)->int {
    return i*i; // square each element
  });
// for loop
hpx::parallel::for_each(hpx::parallel::par,
  v.begin(), v.end(), [](int& i){
    hpx::cout << i << hpx::endl;
  });
```

HPX

Distributed computing

```
int task() { hpx::cout << "Do work" << hpx::endl; }
HPX_PLAIN_ACTION(task);
int main(int argc,char **argv) {
  std::vector<hpx::id_type> localities = hpx::
      find_all_localities();
  for(auto i=localities.begin();i != localities.end()
      ;++i) {
    auto locality = *i;
    task_action ta;
    auto f = hpx::async(ta,locality);
    f.wait();
  } return 0; }
```

HPX

Partitioned vectors, vectors that span multiple nodes (localities).

```
...
#include <hpx/include/partitioned_vector.hpp>
HPX_REGISTER_PARTITIONED_VECTOR(int);
int main(int argc,char **argv) {
  const int size = 100;
  std::vector<hpx::id_type> localities = hpx::find_all_localities();
  hpx::partitioned_vector<int> v =
    hpx::partitioned_vector<int>(size, hpx::container_layout(localities));
  auto f = hpx::parallel::transform(hpx::parallel::par(hpx::parallel::task),
    boost::counting_iterator<int>(0),
    boost::counting_iterator<int>(v.size()),
    v.begin(),
    [](auto i)->int { return i; });
  f.wait();
  return 0;
}
```

Summary

- OpenMP
  - Incremental parallelism with pragmas
  - Parallel instructions distinct from code
- MPI
  - Standard way to do distributed computing
  - Library with method calls
- HPX
  - Fully compatible with the C++11/14/17 standard
  - Distributed and thread parallelism