

An Easier Way

In previous articles, we discussed how to do a parallelization by hand. Among the many conclusions we found, we can safely say it was a complicated process. Indeed, it's so complicated that it's an excellent deterrent to keep people from writing parallel programs!

Fortunately, there is an easier way. If you have been reading this column, you should have a basic understanding of the factors that influence parallel performance. Before we get to the "point and click" part of the column, let's take one a look at the simple program in *Listing One*.

If you have not already noticed, this program code performs a simple integration. Examination of the program tells us we have a simple loop:

```
(do 1 i = 1, N)
```

with a reduction operation:

```
(result=result+f*step)
```

that is parallelizable. Let's see how the *flow model* can be used for this loop.

The Hard Way

Can we know how efficient the program will be before it is parallelized (i.e. should we parallelize this program?) Using the formulas presented in our September *Parallel Lines* column, we can estimate the amount of (input) data used (x and $step$ giving 16 bytes) and the amount of (output) data produced ($result$ presenting 8 bytes).

We know time estimation formulas for each data transfer and for the reduction. We also need to

profile the program and measure the network, so we can get times of each loop iteration, latency, and the transfer time of one byte.

After doing this, we can finally solve the equation and see that speedup of the program can range from 4 to 5 times for eight processors. (This number may change slightly for various architectures.)

One question to ask is: What if we changed the number of intervals from 10000 to 100000? We already know network parameters, so a re-profile of the program is not necessary in this case. (Note that in some cases, changing loop bounds may change the time of one iteration. This situation can be true when arrays are used.) So, we need only solve the equation once more for a higher number of iterations. Fortunately, we can see that speedup will increase.

Had enough? We have. This method of estimation requires a lot of time and headaches. Is there another way to make these estimations?

The Easier Way

Ideally, we would like to have a tool that does all the calculating and optimizing automatically. *BERT 77*TM was designed to do just that. At this point, we will use BERT 77 (or BERT for short) to parallelize the program in *Listing One*. You may wish to download and install BERT 77 so you can following along with the rest of the column (see below).

BERT 77 can be described as an automatic parallel compiler front-end for FORTRAN 77 (the dusty deck codes). It works by analyzing FORTRAN 77 source code, identifying parallel portions of the code, *scheduling* those parallel portions of code for maximum efficiency, and producing new (and readable) parallel FORTRAN 77 source code.

The "magic" of BERT is that it uses a machine profile to determine the most efficient scheduling. In some cases, concurrent loops may not be scheduled as parallel because they are inefficient. The scheduling depends on the target

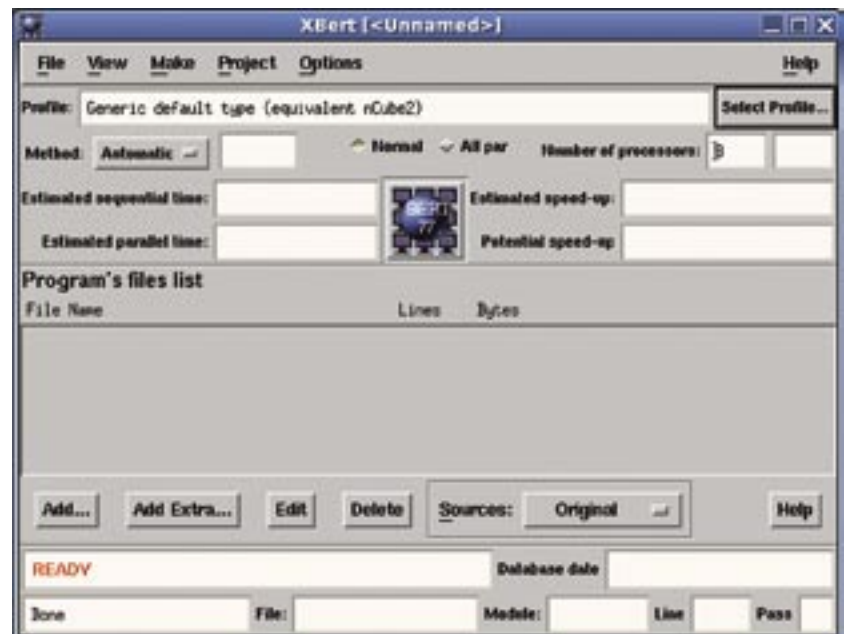


FIGURE ONE BERT 77 User Interface

parallel hardware. In a very real sense, BERT can be considered an optimizing parallel compiler. Just as you need to compile source code for different types of processors, BERT allows you compile your code optimally for different types of parallel computers.

Starting in November 2004, BERT 77 will be available as a fully functional binary version for up to 16 processors. Previously, a free but limited “analysis only” version was available.

You may download BERT from www.basement-supercomputing.com/bert. There is more background material on this site as well.

For now, BERT is available for Red Hat 7.3, 9, and Fedora Core 2. To install BERT simply run (* equals version/system specific information):

```
rpm -i bert-1.05-*.i386.rpm
```

It will install in /opt/bert. See the INSTALL.bert file in this directory for further installation instructions.

[Editor's Note: Pavel and I have

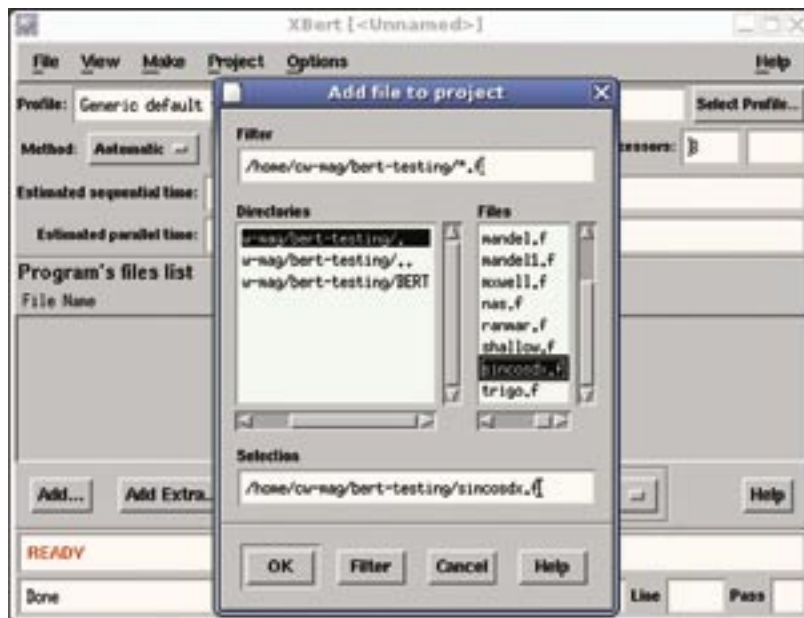


FIGURE TWO File Selection

been involved in developing the BERT project for more than 10 years. Pavel and the BCWs (BERT Cyber Wonks — currently Andrew Bogatov, Vladimir Karpov, Konstantin Buharov) have done most of the heavy lifting. We believe that by providing a fully functional free version we may generate enough interest to continue the project.]

LISTING ONE

A Classic “Dusty Deck” Program

```

program sincosdx
double precision result,f,x,step,func
integer i
real*4 t1,t2
parameter (N= 10000, step=1.d0/float(N))
result=0.d0
do 1 i=1,N
  x=(i-.5d0)*step
  f=func(x)
  result=result+f*step
1  continue
print *, ' result=',result
stop 0
end
double precision function func(x)
double precision x
func=dsin(x)*dcos(1.d0-x)
return
end

```

Once you have BERT installed, you may start the BERT GUI by typing `xbert`. You should see the user interface shown in Figure One. Various aspects of the interface will be presented as we describe how to analyze and then convert the program.

The first thing you need to do is to add a FORTRAN program. The /opt/bert/examples should have a program called `sincosdx.f`.

If it is not there, just copy the program in Listing One.

Once you have a file called `sincosdx`, click on the *Add* button on the main interface screen. You should see a file selection box such as that pictured in Figure Two. Choose the `sincosdx.f` file and click *OK*.

Now that we have the file, we need to select a target parallel environment. This step is accomplished by clicking the *Select Profile* button. A window will open that holds the default profile. (For those interested or who can remember, it is a profile for an nCUBE 2 parallel computer.) The profiles are represented as a tree. By clicking the *Up* and *Down* button, you can navigate the tree. If you select the *Up* button, you will be

taken to the top of the tree. In this example, we will use a PIII/900 system with Myrinet. Scroll to the bottom of the window and select *PIIIDual-900-Linux*. Then click *Down*, highlight *g77*, and click *Down* again. Finally, select *mpi-Myrinet*. You'll notice for this option, the *Down* button is no longer available as you have reached a terminal branch of the options tree. See *Figure Three*. When you click *OK* the *Profile* line should change to *PIIIDual-900; g77; MPI-Myrinet*.

Now it's time for BERT to do some work. When you click the *Make/Information* pull down, BERT will ask you for the name of your project. Enter *sincosdx.xbp* and click *OK*. BERT will then analyze our program. The results should look like those in *Figure Four*.

Some explanation is now in order. The first thing you will notice is there are two times reported for the program: an *estimated* sequential time and an *estimated* parallel time. Before you get too concerned about the absolute correctness of these times, understand that BERT is attempting to make good parallelization decisions.

The time estimates are a function of the target program and the target profile. The relative relationship between these two times is important, not the absolute times themselves.

We see that in this case, both times are quite small and that there is a speedup of 6.4 times on 8 processors. Note that there are two speedups. The first is the *estimated speedup*, which is the best BERT can do given the data dependencies (parallelization inhibitors) of the program. The second is the *potential speedup*, which is based on removing all parallelization inhibitors from your program.

In the case of a simple integration program, these numbers are

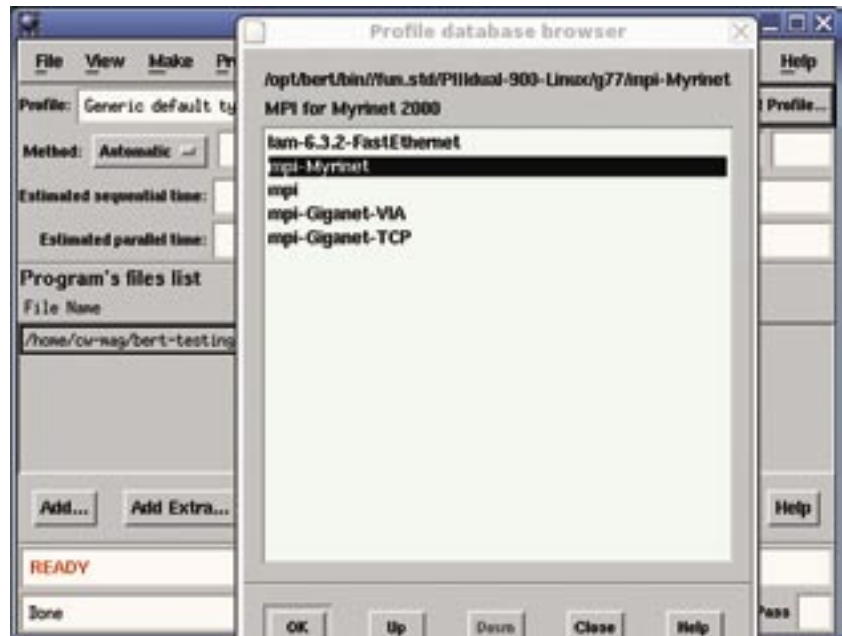


FIGURE THREE Profile Selection

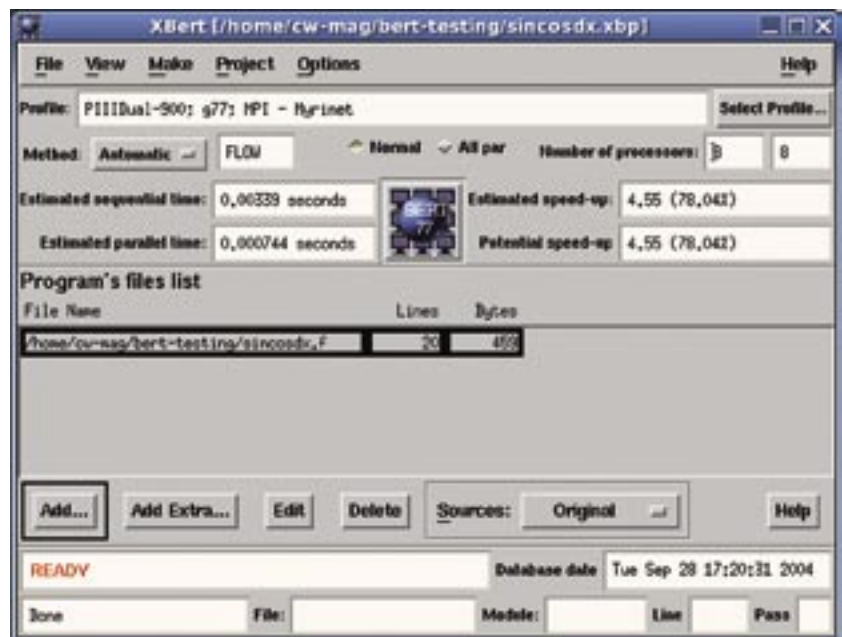


FIGURE FOUR Analysis Results

the same. When evaluating large programs, these numbers can be quite different. Finally, you will see the *Method* box on the left side of the window. This result is the parallelization model

that BERT has selected for the specific program and parallel environment. (See *Parallel Lines* March 2004 for more information.) We can explore the program a bit further. Select the

You can design and optimize codes on your desktop without a parallel computer

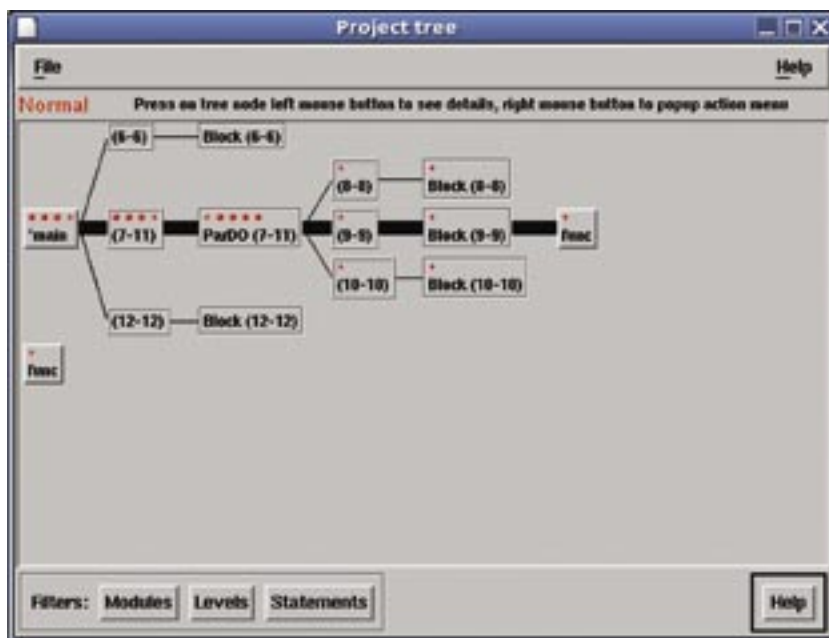


FIGURE FIVE Tree View

The Truth Hurts

It may be surprising to some people, but there is no guarantee of *both* portability and efficiency with most parallel programs. In particular, those written using message passing primitives like MPI or PVM can be particularly troublesome.

Of course, that simple rendering program works on any parallel machine, but can you be sure the parallelization you did for PIII and Myrinet will now work just as efficiently with Opteron and Infiniband?

Because communication primitives and parallelization models are often *explicitly* provided by the programmer, there is no compiler option to “optimize” these parameters.

It is not beyond the realm of possibility that a program written for one parallel environment scales or runs poorly on a new environment that uses faster hardware! Many people often refer to MPI programs as “parallel assembly language” because you are “locking” your program to a specific architecture.

To be fair, not all programs are doomed to suffer this fate. The challenge is to figure out which ones.

View/Project Tree pull down menu. You should see a diagram similar to *Figure Five*. This tree diagram is a visual representation of the program.

Each box in the window represents a section of code. The line numbers are shown in each box. The “computation flow” is shown by the black lines connecting the blocks. The thicker the line, the more com-

putation is taking place in the block. Of particular interest is the block labeled “ParDo” (Parallel Do-loop). This block corresponds to lines 7-11 in the program.

To view the specific lines, right click the mouse on the block and select *view*. A file viewer, like the one pictured in *Figure Six*, will highlight the lines of the parallel do-loop.

In future columns, we will discuss some of the other information BERT provides about these blocks of code.

You may also be wondering what the little red dots mean. These dots are BERT’s way of telling you how efficient the loop will be when run in parallel. In this case, four dots, the maximum, indicates that this loop will be highly efficient. Loops that BERT cannot parallelize, due to loop dependencies, are given blue dots and are called *SeqDo* for Sequential Do-loops.

Sequential do-loops with 3-4 blue dots represent the biggest opportunity for the programmer. These are loops that BERT was unable to convert automatically due to dependencies.

We will learn how to identify these dependencies in a future column. The important point is that these loops represent good parallelization candidates because they will provide a speedup of your code. As a programmer, this is where you should focus your time.

There may be other do-loops that are concurrent but were not parallelized by BERT simply because they would not result in any speedup. Remember, concurrent does not automatically imply parallel execution. If you change the hardware environment, some of those un-parallelized loops may become parallel and vice-versa.

So far, so good. Now let’s try a simple experiment. Edit the program and increase the loop iteration by 10. (e.g. change the $N=10000$ to $N=100000$) You may do this using your favorite text editor or by highlighting the file name in the *Programs File List* and clicking the *Edit* button. A simple text editor will be presented. Don’t forget to save the file after you make the changes.

Click the *Make/Information* pull-

down menu again. You will notice the estimated times have changes along with the estimated speedup. If you have followed along with this column, you will know that this is the expected result.

The amount of work per processor had been increased by a factor of ten so the parallel efficiency has increased (i.e. more time computing and less time communicating). These results are shown in *Figure Seven*.

As you can see, BERT allows you play with some of the variables that effect the parallelization. In the above example, we examined what happens when you change the number of iterations. We could also try adding/reducing processors to see how it effects speedup.

It's also possible, if the profiles are available, to compare various processors, compilers, communication libraries (MPI or PVM), interconnects, even motherboards. As an example, we have seen cases where BERT will choose one parallelization model when using PVM and another when using MPI.

Think about this result for a moment. BERT created two entirely different parallel programs based the choice of message passing library. Not only would this choice difficult for a parallel programmer to determine, but it also makes you wonder how optimal your codes really are.

One of the things that makes BERT particularly interesting is that you do not need a parallel computer to develop efficient parallel programs! It's possible to use BERT to design a parallel algorithm for a specific parallel computing environment without the need for an edit/compile/run cycle.

Indeed, you can design and optimize codes on your desktop without a parallel computer.

Of course, testing your code is

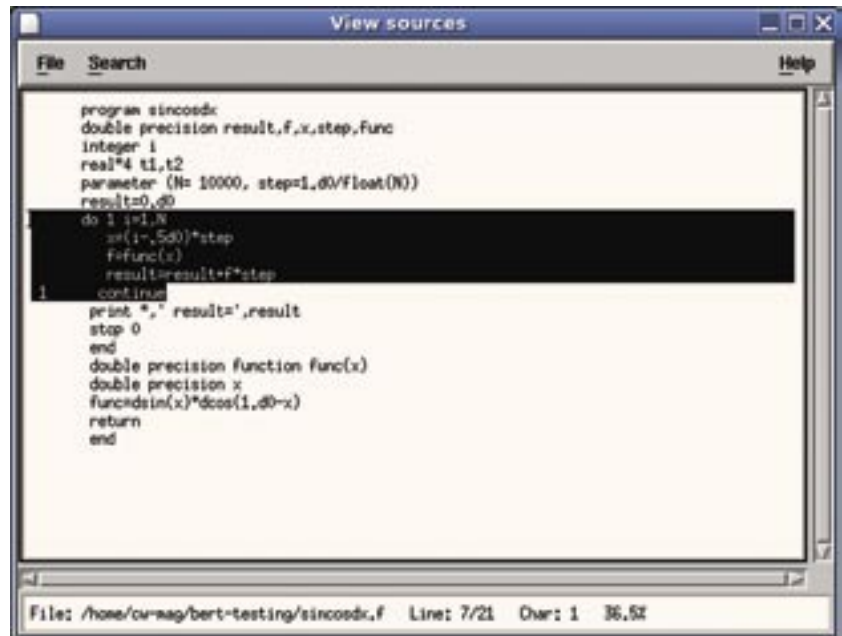


FIGURE SIX Highlighted Parallel Code Block

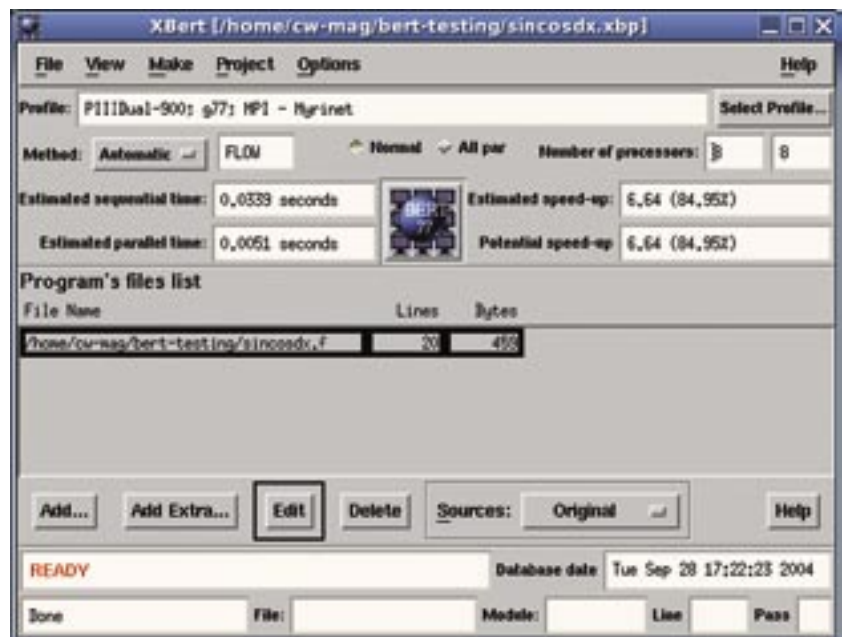


FIGURE SEVEN New Analysis Results

important, but BERT will allow you to make better decisions when you design a code and let you focus your effort where it makes a difference.

We will stop here for this month. We have really just touched on how to use BERT.

Next month we will explore other aspects of the interface and produce some parallel code. Until

then, you can certainly play with the BERT 77 package. And remember, have fun.

Pavel Telegin is currently Department Head of Programming Technology and Methods in the Joint Supercomputer Center, Moscow, Russia. He can be reached at ptelegin@jssc.ru. Douglas Eadline can be reached at deadline@clusterworld.com.