

An Easier Way Part II

In our last installment, we showed how to use the BERT 77™ tool to analyze a simple Fortran code. If you missed last month's article, you can download BERT from www.basement-supercomputing.com/bert. We were able to see how a detailed analysis can be handled with a few mouse clicks. As we will see, it is not always that easy, but for now we can continue using BERT 77 to analyze the simple program we were using last month (see *Listing One*).

As you may recall, BERT uses a specific machine profile when it does a conversion. The machine profile is necessary because the ratio of communication to computation determines the efficiency of parallel execution. As we have emphasized in the past, *what may be parallel for some may not be parallel for all*.

With all this in mind, let's probe BERT a bit further. Before we look at source code generation, however, let's play with a few other parameters that are available in the GUI.

Just Playing Around

BERT gives us the ability to play with some parameters that would normally require a lot of time and effort. For instance, we can change program parameters, numbers of processors, message passing API, compiler, and even the hardware.

Let's take a look at what happens if we change the number of processors. *Figure One* shows the state of our conversion at the end of last month's article. Recall that we had changed the loop count from 10,000 to 100,000 to increase the amount of work on each node. BERT

told us that the speed-up would increase as a result (from 4.55 to 6.64 on eight processors).

The next logical question to ask is what happens if the number of processors is increased?

This task is easily done with BERT. First notice that there are two windows next to the *Number of Processors*: in the GUI. The first window is for setting a new number of processors. The second window is for the current data shown in the interface. In this way, there will be no confusion as to what the current analysis data represent.

To change the number of processors, type 16 in the first window and hit *Enter*. Note, you must hit *Enter* or the old values will be used. You will be reminded that changes do not take effect until you re-make the information. To do this, select the *Make/Information* menu item. BERT has now determined that our

program provides a speed-up of 12.03 on 16 processors. *Figure Two* illustrates these results.

If eight processors is good and sixteen processors is better, then thirty-two processors must be the best (or maybe sixty-four processors). If we were to do the analysis, we would get the results shown in *Table One*.

The first thing to notice is that adding more processors does not give indefinite speed-up. This result assures us that BERT understands Amdahl's law. The second thing to notice is that the best efficiency is around eight processors. The *Estimated speed-up* window in the BERT GUI shows both speed-up and gain, but not efficiency. See the sidebar *Speedup and Efficiency* for definitions of how these numbers are calculated. We are also not limited to the powers of two for the number of processors. You

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

```

may try different numbers of processors near eight and see if the efficiency increases or decreases. Another interesting point is that while the program is most efficient at eight processors, it is fastest at sixteen processors. One could say that more than sixteen will make the program faster, but clearly not by much. The huge drop in efficiency tells us that anything above sixteen processors is probably a waste of resources.

Change Partners

Now that we have been able to verify that BERT “understands” some of the basic limitations of parallel computing, let’s take a look at what happens if we change the cluster we are using. *Figure Three* illustrates the results a different interconnect on the same eight processors we used in *Figure One*. In this case we changed the profile information to use LAM/MPI 6.3.2 over Fast Ethernet. The first thing we notice is that the estimated parallel time is larger resulting in a subsequent decrease in speed-up. Specifically the speed-up decreases from 6.64 using Myrinet to 5.29 using Fast Ethernet. This result is as expected -- a faster network is always better or as good as a slow network. But *how much better* is a function of the application you are using. BERT can be very useful in making hardware assessments.

Where is the Code

Remember those days/weeks/months you took writing the parallel code. Well, BERT is a bit faster. To write parallel code, select *Make/Parallel Sources*. You will see the progress at the bottom of the GUI. Note that all the conversion information is the same as before. In this case, we are using the *PIIIDual-900; g77; MPI - Myrinet* profile.



FIGURE ONE Analysis results for eight CPUs

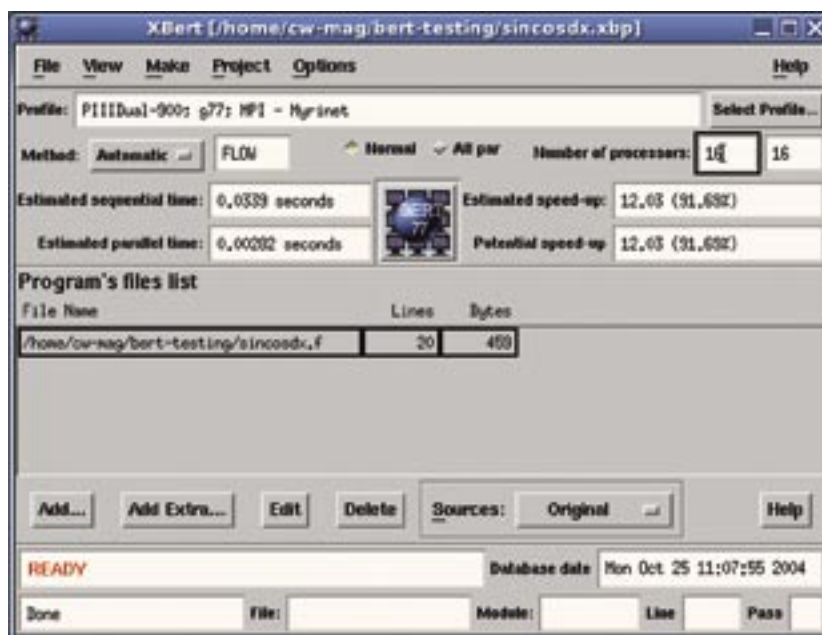


FIGURE TWO New analysis results

BERT will generate the new source code, a Makefile, and a LOAD file for running the code. In our case, the code is located in `working-directory/BERT/sincosdx/parallel`. Other saved projects will place their code in `working-directory/BERT/project/parallel`. One important note, each time BERT writes

new source code, it removes and creates a new *project* directory. If you want to save the code, copy it to another directory.

The files created by BERT for *sincosdx* conversion are shown in *Listing Two*. In addition to the source files, Makefile, and LOAD file, you will see there is a library file called `libBERT.a`. This library file is a

thin layer that maps the BERT parallel calls to the underlying message passing library (MPI or PVM). During the design of BERT it was found that fewer bugs were introduced if we used a single set of BERT parallel calls rather than trying to insert PVM, MPI or some other API directly into the output source code. In addition, if other message passing libraries are used, a new `libBERT.a` is all that is required.

Under the Hood

It can be instructive to see how BERT writes the new code. First, BERT chose the FLOW model as optimum for this conversion (See the *Method* window in *Figure Three*.) In this model, there will be a master program and a number of worker programs. If we look in the file `master00.f` we see the following:

```
result=0.d0
call \
bert_exec_level_7(result,f,x,i)
1 continue
```

Compare this with the code in *Listing One*. Note the preservation of

the loop context. This function implements data communications and invokes parallel execution of the DO loop. You can trace it into the support file `mstsproo.f` where it is basically a master-worker algorithm. It used two main routines, `bert_send_level_7` and `bert_`

`rec_level_7`. These routines eventually call `bert_net_send` and `bert_net_receive` which are calls to the `libBERT.a` library.

If you examine `worker00.f`, you will see the code in *Listing Three*. Note the corresponding receives in to the sends in `mstrspr00.f`. Also you will notice the actual “work” of the loop being done (`f=func(x)`).

Finally, back in `mstspr.f` you can see the following reduction sum in the lines:

```
call bert_net_receive( \
    result_bert, 1, 0, 8 )
result=result+result_bert
```

To explore the source code further, you may wish to set the *Project/BERT Options: Text generation* option to *Insert BERT comments* and see how BERT explains the code.

Making the Code

If everything has gone according to plan, you should be able to type “make” and the program will build. In general, any problems you have

TABLE ONE

BERT 77 results for different number of processors

PROCESSORS	ESTIMATED PARALLEL TIME	SPEED-UP	EFFICIENCY
4	.0114	2.97	74%
8	.0051	6.64	83%
16	.00028	12.03	75%
32	.00024	15.11	47%
64	.00024	15.11	47%

LISTING TWO

Files produced by BERT

conv.inc	LOAD	masteroo.f	mstsproo.f	split.inc	workeroo.f
error.inc	Makefile	master.prj	net.inc	support.f	worker.prj
libBERT.a	Makefile.cfg	memory.inc	packet.inc	warning.inc	wrksproo.f

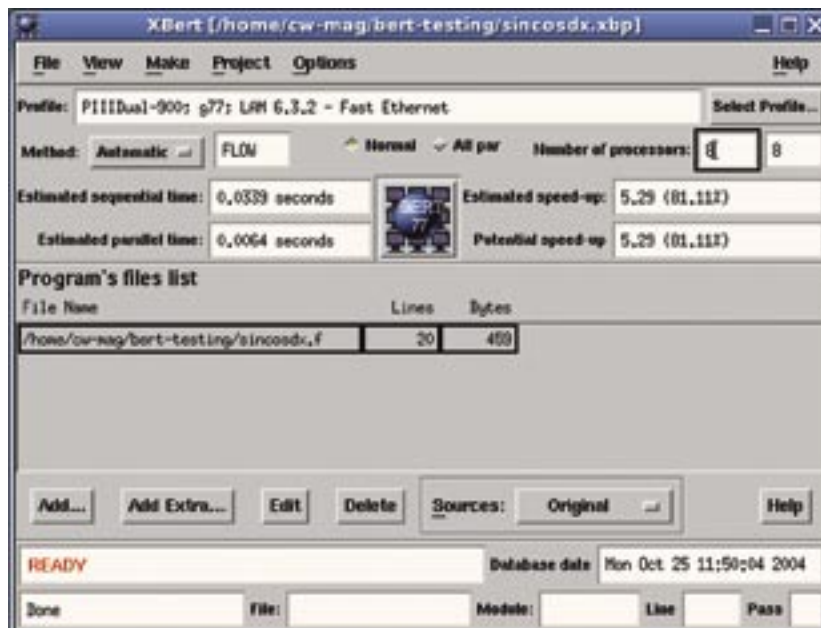


FIGURE THREE Using a different machine profile

Speedup and Efficiency

The terms speedup and efficiency are often used when talking about clusters. The following definitions will be used in this column.

- **speedup** = sequential run-time/parallel run-time program
- **efficiency** = speedup/number of processors
- **Gain** = (sequential run-time - parallel run-time)/Sequential run-time

Speedup shows how much faster a parallel version of the program is compared to a sequential version. *Efficiency* shows, in percent, how well the program scales on a cluster. *Gain* is the percent improvement in run-time (the closer to 100 percent the better).

LISTING THREE

BERT Worker code

```

call bert_net_receive( i_bert_lst, 1, 0, 4 )
call bert_net_receive( i_bert_cur, 1, 0, 4 )
call bert_net_receive( i_bert_low, 1, 0, 4 )
call bert_net_receive( i_bert_up, 1, 0, 4 )
call bert_net_receive( i_bert_stp, 1, 0, 4 )
    result=0.D0
    call bert_stop_receive()
    do i = i_bert_low, i_bert_up, (1)
        x=(i-.5d0)*step
        f=func(x)
        result=result+f*step
1    continue
        enddo
    dowhile ( bert_start_send(bert_master_
        node).ne.bert_normal_state
*    )
    enddo
    call bert_net_send( bert_data_packet, 1, 0, 4 )
    call bert_net_send( 7, 1, 0, 4 )
    call bert_net_send( i_bert_lst, 1, 0, 4 )
    call bert_net_send( i_bert_cur, 1, 0, 4 )
    call bert_net_send( i_bert_low, 1, 0, 4 )
    call bert_net_send( i_bert_up, 1, 0, 4 )
    call bert_net_send( i_bert_stp, 1, 0, 4 )

```

will probably have to do with linking and are similar to building any other code. Note, BERT uses the standard MPI wrapper functions, so most of these problems should be avoided.

If the code builds, you will two binaries: `master` and `worker`. To run the codes a script called `LOAD` is provided. Consult the BERT docu-

mentation or the `LOAD` script itself for more information.

Changing Things

If you look closely at the Makefile, you will see, in this case we are using LAM/MPI and `g77`. Note that changes in compiler type may effect the conversion, so a profile for your specific compiler is important.

BERT allows you to change almost all the arguments for the compiler and linker used in the Makefile. If you click on *Project/BERT Options*, then click on *Makefile Options* on the right side of the window, you will see the window shown in *Figure Four*. As you can see, there are boxes to enter/add options for all aspects of the make process. These options are saved as part of the project file, so once they are set they will be used in future Makefiles. This feature is why BERT options appear under the *Project* menu and not the *GUI Options* menu.

While we are looking at BERT options, let's examine some of the other general options you may wish to change. If you click on *Syntax and Parallel Analysis*, you will see the screen shown in *Figure Five*. The most important option here is the number of continuation lines and line length. You can also specify if you want *loops and blocks* parallelized or just *blocks* or *loops*. In some cases, you may not want BERT to parallelize blocks as it may result in an unbalanced program. Finally, there is an option to reorder I/O statements. If, for instance, you are just printing out status messages, BERT will reorder these to increase performance. The messages may not make any sense after this re-order, however.

We will take a look at some of the other options in the future. For now these are the basic options that will help you get started.

Helping BERT

BERT is not without limitations. When evaluating code, there are some instances where BERT cannot determine the dependencies in a loop and therefore cannot automatically parallelize loops. Indeed, the loop may use a large amount of processing time and it may be a good candidate for parallelization.

For this situation, there is a set of “pragmas” to give BERT hints and allow you to control the parallelization. These pragmas can also be used for development of new parallel code to make program initially parallel. We will discuss this in future columns. For now, have fun playing with BERT.

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

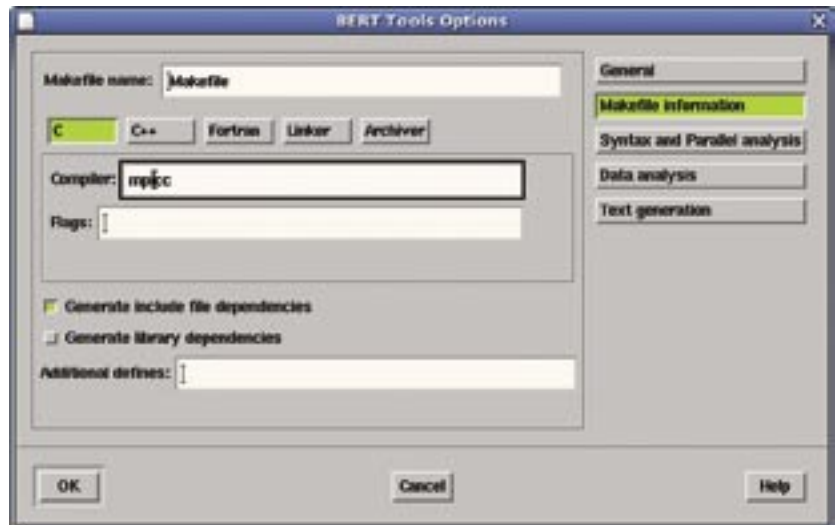


FIGURE FOUR Makefile options

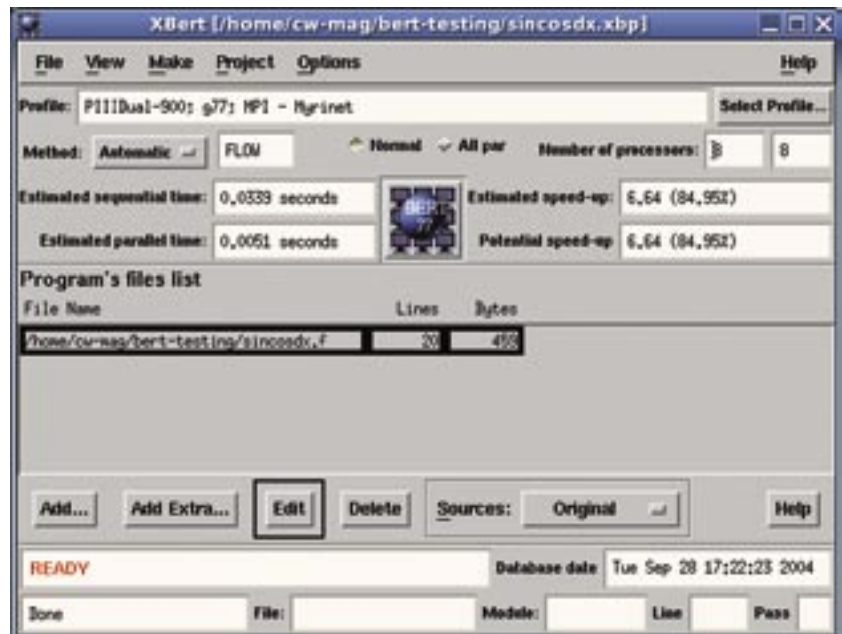


FIGURE FIVE Syntax and Parallel Analysis