Case Report ■

A High Productivity/Low Maintenance Approach to High-performance Computation for Biomedicine: Four Case Studies

NICHOLAS CARRIERO, PHD, MICHAEL V. OSIER, PHD, KEI-HOI CHEUNG, PHD, PERRY L. MILLER, MD, PHD, MARK GERSTEIN, PHD, HONGYU ZHAO, PHD, BAOLIN WU, PHD, SCOTT RIFKIN, PHD, JOSEPH CHANG, PHD, HEPING ZHANG, PHD, KEVIN WHITE, PHD, KENNETH WILLIAMS, PHD, MARTIN SCHULTZ, PHD

Abstract The rapid advances in high-throughput biotechnologies such as DNA microarrays and mass spectrometry have generated vast amounts of data ranging from gene expression to proteomics data. The large size and complexity involved in analyzing such data demand a significant amount of computing power. High-performance computation (HPC) is an attractive and increasingly affordable approach to help meet this challenge. There is a spectrum of techniques that can be used to achieve computational speedup with varying degrees of impact in terms of how drastic a change is required to allow the software to run on an HPC platform. This paper describes a high-productivity/low-maintenance (HP/LM) approach to HPC that is based on establishing a collaborative relationship between the bioinformaticist and HPC expert that respects the former's codes and minimizes the latter's efforts. The goal of this approach is to make it easy for bioinformatics researchers to continue to make iterative refinements to their programs, while still being able to take advantage of HPC. The paper describes our experience applying these HP/LM techniques in four bioinformatics case studies: (1) genome-wide sequence comparison using Blast, (2) identification of biomarkers based on statistical analysis of large mass spectrometry data sets, (3) complex genetic analysis involving ordinal phenotypes, (4) large-scale assessment of the effect of possible errors in analyzing microarray data. The case studies illustrate how the HP/LM approach can be applied to a range of representative bioinformatics applications and how the approach can lead to significant speedup of computationally intensive bioinformatics applications, while making only modest modifications to the programs themselves.

■ **J Am Med Inform Assoc.** 2005;12:90–98. DOI 10.1197/jamia.M1571.

This paper describes four case studies that apply high-performance computation (HPC) to different problems involving bioinformatics. In particular, the paper describes a high-productivity/low-maintenance (HP/LM) approach to bringing the potential power of HPC to bioinformatics. The goal of this approach is to allow bioinformatics researchers to reap the benefits of HPC without having to make major sacrifices in the ability to pursue their research, which frequently involves a process of iterative algorithmic exploration and refinement over time. HP reflects an emphasis on minimizing the initial HPC deployment effort, whereas LM stresses the importance of reducing the ongoing work needed to continue to realize the benefits of HPC during the process of iterative refinement.

As discussed later in the paper, a variety of techniques can be used to achieve HPC. One major technique involves parallel computation in which different parts of a computationally intensive program run simultaneously on multiple computer processors. Parallel computation can be performed on special-purpose parallel machines or on clusters of conventional workstations or computers connected via a computer network.

Researchers have used parallel computation extensively to enhance the performance of many computationally intensive applications in biomedicine. One potential risk in porting a program to run on a parallel machine is that the resulting parallel program may be very different from the original program developed by the biology/bioinformatics research team. In the extreme case,

1. The original program may be completely rewritten in a different programming language for efficiency purposes. For example, it might be translated from the interpretive Perl language to the compiled C language.

Affiliations of the authors: Department of Computer Science (NC, MS), Center for Medical Informatics (MVO, K-HC, PLM), Department of Genetics (HZhao, KWh); Department of Molecular Biophysics and Biochemistry (MG, KWi); Department of Molecular, Cellular, and Developmental Biology (PLM); Department of Statistics (JC); Department of Epidemiology and Public Health (HZhao, BW, HZhan); Department of Ecology and Evolutionary Biology (SR); and W. M. Keck Biotechnology Resource Laboratory (KWi), Yale University, New Haven, CT.

Supported in part by National Institutes of Health (NIH) grant K25 HG02378 from the National Human Genome Research Institute; NIH grants T15 LM07056 and P20 LM07253 from the National Library of Medicine; NIH contract N01-NV-28186 from the National Heart, Lung, and Blood Institute; NIH grant U24 DK58776 from the National Institute of Diabetes and Digestive and Kidney Diseases; and by National Science Foundation (NSF) grant DBI-0135442.

Correspondence and reprints: Nicholas Carriero, Department of Computer Science, Yale University, New Haven, CT 06520-8285; e-mail: <carriero-nicholas@yale.edu>.

Received for publication: 03/09/04; accepted for publication: 08/05/04.

2. The program code may be restructured in fundamental ways to make it more efficient, for example, to take advantage of specialized algorithmic programming techniques more familiar to computer scientists than to biological application developers.
3. Parallel programming constructs may be tightly integrated into the code at various levels to take maximum advantage of the potential for parallel execution.

The above would *not* be examples of an HP/LM approach to HPC. Once all these changes, refinements, and transformations have been made to the original code, it would be difficult for the biology/bioinformatics team to work with that code in a very flexible, incremental fashion. It would now be much more difficult to refine their approach and their ideas iteratively and to try many different variants of a solution. They would have to make such incremental changes to the new, parallelized version of the code. Alternatively, if they continued to develop and refine their original program, any changes would need to be replicated in the parallel version to obtain the advantages of HPC. Both of these solutions are potentially awkward and inhibiting.

As a result, this HM approach to HPC risks "freezing the program in concrete" in a way that, in practice, may make it very difficult for the code to evolve. In addition, the level of effort required effectively prohibits broad deployment of this approach. Nevertheless, this approach to HPC may be highly desirable for extremely computationally intensive applications, particularly if the underlying program has become mature and iterative refinement is not a major goal.

We believe, however, that there is a major potential benefit to developing and refining an alternative HP/LM approach that will help enfranchise many more bioinformatics researchers and applications and help them harness the potential power of HPC. Underlying this HP/LM approach are several fundamental principles.

1. A major principle of the HP/LM approach is to make as few changes to the original code as possible. In fact, if it is easy to structure parallel computation around the original code without changing it at all, and still reap the benefits of HPC, that would be most desirable.
2. The goal of the HP/LM approach is not to reap the maximum efficiency benefits in making changes to a program. Rather, *the goal is to remove compute time as a rate-limiting step in biology/bioinformatics research*. Thus, if a computation takes a month to run and can be reduced to ten hours, that may be sufficient if the rate-limiting step now becomes performing the laboratory research and/or analyzing and organizing the data. It may not be necessary, for example, to invest more time and energy to reduce the computation to one hour. As a result, one might decide to leave a program as a whole in an interpretive language and only optimize particular highly computationally intensive modules.
3. HPC is not a synonym for parallel computation. For some programs, parallel computation may not even be required to achieve HPC. As described below, an important first step in approaching a new program presented for porting to HPC is to inspect the code and see whether simple, local programming changes might greatly speed up the program. Typically, this is the case. Indeed, it may be that relatively simple, local changes are all that are needed to remove the computation time of that program from being a rate-limited step in the research project as a whole. In that case, parallelization may be completely unnecessary.

Taken together, these principles can foster a collaborative effort well within the comfort zones of both the bioinformaticist and the HPC expert. This paper describes four case studies that illustrate these principles at work.

Goal of this Paper

The goal of this paper is to help facilitate the widespread use of HPC by bioinformatics researchers. Bioinformatics is increasingly tackling compute-intensive problems that require HPC, and HPC is increasingly affordable. We believe that the HP/LM approach represents a philosophy and an approach to the use of HPC that has the potential to make HPC much more accessible to bioinformaticists in the context of iterative algorithmic research. This approach, of course, complements fundamental research to develop new algorithms to perform the underlying computations much more efficiently. To help make the HP/LM approach more concrete, the paper describes our experience applying these principles in four representative case studies, with specific examples of how HPC techniques were used for each. In addition, we discuss various ancillary issues that arose and that were important to achieving success but are not typically considered to be part of HPC per se. The broader goal of this work is to explore ways to make it easier to bring HPC to bioinformatics and thereby to make it possible to use HPC in many more bioinformatics applications than might otherwise be the case.

Background

The field of parallel computation has been a robust area for research and development within the field of computer science for many years. Much of the early work in this area involved the use of special-purpose parallel machines and continues to the present day. Examples include the iPSC/2,[1] CM-5,[2] ASCI Red,[3] and Blue Gene.[4] A great deal of recent work involves parallel computing on networks or clusters of "off-the-shelf" desktop computers and servers. Good parallel programming tools are essential. Examples include message-passing systems PVM (parallel virtual machine)[5] and MPI (message passing interface)[6] and virtual shared memory coordination languages such as Linda.[7] There has been a wide range of computationally intensive computations within biomedicine in which these technologies have been successfully applied, including linkage analysis,[8] sequence comparison using Blast,[9] and FASTA search.[10]

A recent trend within the biological sciences has been the development of a range of high-throughput technologies that are starting to produce massive quantities of data on a scale that far exceeds previous data produced by biological research. These technologies, for example, include the use of microarray technology to analyze gene expression[11,12] and the use of mass spectrometer–based techniques to analyze protein expression.[13,14]

As a result of technologies such as these, a rapidly growing number of biomedical researchers need to analyze data in quantities that are larger by several orders of magnitude than they have analyzed previously. Many of these researchers therefore have to confront problems that involve very computationally intensive analyses. In addition, many of the algorithms and approaches to analyze these massive amounts of data are still in the very early stages of development. Bioinformatics researchers are exploring many different approaches and need to be able to do so in a very flexible, incremental, evolutionary process.

Thus, there is a need for a new type of collaboration between computer scientists and biomedical researchers, one that will bring HPC to bear on the many evolving challenges in a way that accommodates the fluid nature of the underlying bioinformatics research. This is the context in which we are exploring the HP/LM approach to biomedical HPC described in this study.

A number of techniques are used to improve the performance of the codes discussed in these case studies.[15,16] We make no claim of novelty with respect to these techniques per se. For example, one class of techniques has its roots in compiler optimization: strength reduction, common subexpression elimination, code hoisting, loop reordering, etc.[17] Many times, compilers (or interpreters) lack the information necessary to apply these optimizations automatically safely, but it is straightforward for a human to do so manually. For the most part, these techniques (and many others) are well known to experienced programmers. Of equal importance, experienced programmers know when it is appropriate to use these techniques and how to apply these techniques in an unobtrusive way. The question of concern here is how to bridge this expertise gap so that this knowledge will be brought to bear in the service of bioinformaticists and their codes, with a minimum of disruption to the base code and while minimizing the efforts of the experts (bioinformaticists, biologists, and computer scientists).

Four Case Studies

Using the Blast Sequence Comparison Program on a Massive Scale

The first case study involves a project that requires performing biological sequence comparisons on a massive scale, using the widely used Blast sequence comparison program.[18] An example of this type of computation might be to compare a database of 300,000 known protein sequences with an entire genome, which might contain hundreds of millions of nucleotide base pairs, or more. Another example might involve comparing multiple genomes with one another. Large Blast runs, on the order of 10^7–10^8 symbols (nucleotides or amino acids) against aggregate data sets in the range of 10^8–10^9 symbols are playing an increasingly important role in some genomic and proteomic analyses.

An example problem being explored at Yale involved a 500 million amino acid aggregate data set (from seven source data sets) compared with itself. We initially investigated three HPC strategies for delivering enough compute power to carry out these computations in a reasonable amount of time. In keeping with our rate-limiting step goal, our aim was to reduce turnaround time from the better part of a year (definitely rate limiting) to one or two weeks (considerably less than the time needed to propose such a problem, gather data sets, draw conclusions, and author a paper[19]).

1. The first strategy involved using a special-purpose, commercially available parallel package for performing Blast searches.
2. The second strategy involved exploring the use of a parallelized version of the Blast code, in which parts of an individual sequence comparison could be performed in parallel.
3. The third strategy, which we ultimately adopted, was to use the standard (sequential) Blast package and to achieve a simple level of parallelization using Python scripts. We settled on this strategy because it was by far the simplest, yet was effective for the project at hand. It also required no changes to the Blast program and was therefore firmly in accordance with our HP/LM philosophy.

The Blast runs of interest were easy to decompose into manageable computations that could be run independently on separate processors, the available resources are essentially dedicated, and data could be made available via a network file server. As a result, it was straightforward to implement the scripting approach, which required creating scripts to do the following:

1. Split the target file into chunks, each of which defined separate sequence comparison tasks.
2. Detect idle cluster nodes.
3. Launch new Blast jobs to idle nodes, working through the collection of tasks. A special "driving script" was written to look repeatedly for free nodes upon which to launch new tasks automatically.
4. When all the tasks have been processed, concatenate the output from individual runs to form the final output.

These scripts were all easily realized in Python. None was more than a page. A total of only 125 lines of code was required in all. Using these scripts, we performed several runs, which formerly would have taken several months each, in ten days to two weeks each. The runs were performed using a 12-node parallel computer cluster, in which each node was a dual-processor machine with 2.4 GHz per processor and 2 GB of main memory per node. In summary, for this problem, the HP/LM scripting approach was simple, involved no alteration to the basic Blast system, was robust, and accomplished the goals with reasonable efficiency and modest deployment effort.

Taking this scripting approach and leaving the Blast program unchanged did raise a number of ancillary problems that needed to be solved. These problems would not normally be considered to be issues involving HPC per se, but they do serve to illustrate some of the pragmatic problems that arise when adapting an application for HPC. For this Blast application, the following issues arose:

1. *The need for automatic data cleaning*. One major problem that arose concerned the format of the input data. To run successfully with Blast, the data needed to be in a specific format (the FASTA format). If some of the data were not strictly in this format, the Blast comparison would fail. Sometimes it would fail in a readily detectable way. Sometimes it would fail in a way that was not easy to detect short of comparing the results with a sequential Blast run on the same data. As a result, a data screening script needed to be written that would ensure that the data were formatted correctly before the Blast system was run.
2. *Blast error detection*. Sometimes the Blast program failed for other reasons, for example, due to incompatibilities

between how Blast and Linux use memory. Such errors could result in the Blast program stopping without completing its analysis, but with no indication that an error had occurred. To detect such situations, the scripts needed to inspect the output file to ensure that the full analysis had been performed. This could be performed automatically. We also reduced the Blast task size so that such errors were less likely to occur. Note that this problem could, of course, occur with just a single Blast run, but, in that case, it would naturally be easier to notice. When the Blast runs were performed on a massive scale, an automated tool to detect when the problem has occurred was needed.

Thus, this case illustrates that a large computation could be approached using HPC in a quite straightforward fashion, with no change to the bioinformatics program being used. This general approach clearly has potential use in many applications beyond Blast. Some specialized error checking was required to handle situations that were presumably caught by manual inspection when Blast runs are performed on a much less massive scale.

Analysis of Large Mass Spectrometry Data Sets

The second case study involves a biostatistics project that is exploring approaches to cancer classification using mass spectrometry (MS) data sets, attempting to identify biomarkers in serum to distinguish between cancer and normal samples.[20,21] In this application, the MS data sets each involve a matrix in which the rows were mass/charge ratios observed by MS and the columns correspond to anonymized patient serum samples. Each cell in the matrix contains the observed intensity for the corresponding mass/charge ratio and sample. A second vector indicates each sample's cancer status.

The project used the RandomForest (RF) algorithm developed by Breiman[22] for classification analysis. Previous work has shown good performance using this algorithm.[23] Briefly, the RF algorithm uses bootstrapping from the original data to create multiple pseudo data sets. These are then used to try to identify a modest subset of the features that best split the data set into normal versus disease categories. This is done by randomly selecting a different subset of features for each pseudo data set and seeing how well that subset of features works. RF combines two useful ideas in machine-learning techniques: bagging and random feature selection. Bagging stands for bootstrap aggregating, which uses resampling to produce pseudo replicates to improve predictive accuracy. By using random feature selections, predictive accuracy can be significantly improved. As the resampling step can be easily parallelized, this algorithm provides an excellent example that demonstrates the power of parallel computing in proteomics research.

This RF code passed through two iterations of HPC refinement and at the same time evolved significantly in response to developments in the code owners' research program. The first phase involved modest local optimization of the code by elimination of common subexpressions from nested loops, in a similar fashion as discussed in the next case. The changes in the second phase focused on four issues: data structures, numeric stability, parallelization, and random number generation.

The main data structure of the code stores a matrix of input data sorted by rows. As the algorithm unfolds, the matrix is partitioned into subsets of columns and the data within the subsets are shuffled to preserve sorted order in the rows. The intent here was to avoid the cost of a full sort for each row fragment with each iteration of the code. A careful review, however, revealed that significantly more time was spent in carefully maintaining the sort order of the whole data set than would be required to sort on the fly those portions of the data set actually used. The code was correspondingly restructured using a standard Unix library sort routine, and in the process the program was simplified.

The code also exhibited some problems when we attempted to use compiler optimizations in that the output differed. This problem was ultimately traced to conditional constructs using direct floating point numeric comparisons. These were replaced by comparisons that ignored differences within a small epsilon, which allowed the program in turn to accommodate floating-point round-off error. Once these changes were made, compiler optimizations could be enabled without altering the output.

These changes (along with improvements in the first phase) led to an approximately 18-fold reduction in execution time. This is a rough estimate because, as indicated above, the base code was evolving for other reasons, making it difficult to compare directly with the initial version.

Finally, the generation of each RF tree is, in principle, an independent computation, which allowed a straightforward implementation of parallelization, namely, to farm out that generation of trees to a collection of worker processes. To realize this strategy with minimal impact on the code, we made use of a parallelization "idiom" (described in detail with examples[24]) that is based on creating multiple processes that are essentially replicas of the initial process, all executing in loose synchrony until they reach an inner loop where they cooperatively work to define and execute a partition of the work that is to be done in the body of the loop. In abstract terms, a sequential program with the following general structure:

```
setup()
loop {
  loopWork()
}
cleanup()
```

becomes a parallel program with the following logic at each of multiple simultaneous processors:

```
setup()
loop {
  if ownThisWork()  logResult(loopWork())
}
mergeResults()
cleanup()
```

ownThisWork( ) is a bit of coordination code (written in a variant of Linda[7,25]) that affects the disjoint partitioning. When multiple instances of the above are executing concurrently, ownThisWork( ) ensures that one and only one instance evaluates a given iteration of the loop. The net result is that the base code need only be modified in a few places with a few lines of code and a coordination module linked to the final executable to realize a parallel version. We also developed a sequential version of the coordination module so that the same source code could be run as a "traditional" sequential executable in an environment lacking multiple processors.

The only other change involved restructuring the code's interaction with a random number generator to ensure identical output from sequential and parallel runs. One hundred eighty lines of code (approximately 20 lines in the base code, the rest in the coordination modules) were required to accomplish all aspects of this parallelization, half of which is easily recyclable "boilerplate." The parallel version ran about 11 times faster on 20 processors, for a net reduction in run time of a factor of about 200 from the initial run time.

Genetic Analysis

The next case study involves a program that implements genetic analysis of ordinal phenotypes. Numerous human diseases and health conditions are recorded in ordinal scales including various stages of cancer and severity of psychiatric disorders. Genes and gene-environment interactions underlie many of these conditions. Statistical models and software are well established when the disease is diagnosed as "yes" or "no" or reflected by a continuous measure such as the relationship between hypertension and blood pressure. However, little attention has been paid to the genetic analysis of ordinal phenotypes. Zhang et al.[26] outline a latent variable model to assess whether an ordinal trait aggregates within a family and whether it has a major gene effect. Due to the complex nature of family structure and the complexity of the latent variable, the evaluation of the likelihood function involves a hierarchical summation and is computationally intensive. In addition, to evaluate the validity of the proportional odds model, a series of simulation studies can be performed for which increased computational performance would also be very helpful.

A first step in considering how best to port a program to HPC involves profiling the program to see where it spends most of its compute time. If highly computationally intensive portions of code are identified (e.g., inside nested loops), one then examines those sections of the program for potential optimizations. In this case, we identified three classes of changes to the base code to improve performance.

1. Profiling indicated the code spent a significant amount of time computing integer powers of 3 using the pow( ) function. That function is really meant for computing real powers of real numbers and is therefore much more involved than what was actually needed. In fact, the needs of this program could be served using lookup within a simple table of the powers of 3. Thus, a computationally intensive function invocation like pow(3, i) could be replaced with a simple table lookup, e.g., pow3tab[i]. (Note: The next change makes this change less important, as it significantly reduces the need for powers of 3 in the first place.)
2. Looking into why powers of 3 were needed, however, it became clear that the main use was to convert an index variable into its base 3 digits. Rather than recompute the expansion every iteration of a nested loop, it is much more efficient just to add 1 to the previous base 3 expansion, using the rules of base 3 arithmetic. In the original program, the following code converts j, the outer loop index variable, to its base 3 representation (stored in the array indexk):

```
for(k = 0; k<MaxDigit; k++) {
   rr = j/pow(3, (MaxDigit-1-k));
   j = j - rr * pow(3, (MaxDigit-1-k));
   indexk[k] = rr;
 }
```

This code was replaced with code that added 1 to the base 3 representation in indexk each iteration, thus avoiding the calculation of pow() entirely:

```
if(j){
   digit = MaxDigit-1;
   while (2 == indexk[digit]) {
      indexk[digit] = 0;
      --digit;
   }
   indexk[digit] += 1;
}
```

3. The third modification involved introducing a temporary variable to hold the value of an expression that is expensive to compute, and moving that computation out of a nested loop, which is possible because the value of the expression does not change during the iterations of the loop. This is a simple technique that can be used to speed up many programs. In this way, the following code

```
for(l = 1;  l < = p+4;  l++) {
   for (k = 1; k < = p+4; k++)
       I22[l] [k] += li[l] * li[k] * (theta1 * exp(Pi1)
                      +(1-theta1) * exp(Pi2));
   I12[l]  += li[0] * li[l] * (theta1 * exp(Pi1)
                + (1-theta1) * exp(Pi2));
   I120[l] = I12[l];
}
```

became

```
cse = (theta1 * exp(Pi1)  +  (1 − theta1) * exp(Pi2));
for(l = 1; l < = p+4; l++) {
   for(k = 1; k < = p+4; k++)
      I22[l][k] += li[l] * li[k] * cse;
   I12[l]  += li[0] * li[l] * cse;
   I120[l] = I12[l];
}
```

These three changes, which involved changing only a few lines of code, resulted in an approximately fivefold reduction in run time. It is worth emphasizing, however, the following:

1. The original program should not be construed as somehow "wrong." It reflects a natural expression of the computation to be done.
2. The expertise to realize the performance implications of these changes is broadly available in one community but not in another.
3. As a result, this is an excellent example of the HP/LM principle in action.

Building on this improved sequential base, we developed a parallel variant of the code. This effort was quite similar to that described in the previous section. The "ownership" technique described above was used, entailing only a few changes to the base code. One hundred fifty lines of code were required to accomplish this parallelization, roughly half of which were "boilerplate" from the previous case.

Both parallel and sequential coordination modules were written that allowed the creation of executables for either environment. A speedup of approximately eightfold was achieved with 24 compute nodes. A limiting factor in parallel performance was Amdahl's law: 190 seconds of a 3,500-second sequential test run (using the optimized sequential code) was spent in code that was not parallelized. As a result, the best time for a perfectly efficient parallel run would be 320 seconds (190 + (3,500 − 190)/24). Our parallel test ran in 450 seconds. Overall, the speedup relative to the original code was approximately 30- to 40-fold for our test cases.

Microarray Analysis

The fourth case study involves a program designed to help assess the effect of possible error in analyzing microarray data. Ever since microarrays were invented as a tool to measure the transcriptional state of a genome, their uses have expanded, but all applications share some common statistical and computational problems. The many thousands of measurements on any single microarray and the millions of measurements comprising a complex multiarray experiment share sources of noise as well as actual biological signal. Exactly how to translate the raw data, approximately 20 pixels worth of fluorescence intensities for each spot, into abundances of mRNA molecules, for instance, is still unknown. Various methods have been proposed to estimate and remove noise,[27,28] but gauging how well these work still awaits better physical models of the experimental process.

To analyze the results of microarray experiments designed to investigate gene expression in the fruit fly,[29] a code was developed "in house" that incorporates a general linear model, specifying sources of signal and noise and estimating their contributions to the overall measurements.[30] Although an R program was available to do this analysis,[31] it lacks the ability to cope with heteroscedastic data. The initial code calculates the relevant averages over various combinations of the expression data and from these computes an analysis of variance. Bootstrapping techniques were used to develop confidence intervals.

The analysis code was written in Python, a language well suited to rapid experimentation and development of algorithms. Python is object oriented and interpreted, making it a rich, but sometimes sluggish, environment in which to work. We augmented the code with a timer facility to obtain profiling data. These data guided a series of changes that fell into two groups: (1) a shift from in-line code to the use of optimized Python extensions for numeric operations and (2) a move to more efficient Python file operations.

The use of optimized extensions (or more generally to the use of standard efficiently coded libraries) is a particularly important part of the HP/LM HPC approach. Code reuse, of which this is an instance, is a desirable goal of computer software systems, but, in practice, there are often impediments for the nonspecialist user. Different vocabularies, data structures, and usage profiles combine to make the integration of good packages a difficult proposition. Easing these integration issues is a topic worthy of systems research. We illustrate here some of these issues in the narrow context of the code at hand. Consider the following code fragment:

```
for f in range(len(effects[e])):
  if len(effects[e].shape) > 1: #if it is a double effect array
    for g in range(len(effects[e][f])):
      if counter[e][f][g] != 0:
        effects[e][f][g] = effects[e][f][g]/counter[e][f][g]
      else:
        effects[e][f][g] = 10000
  else:
    if counter[e][f] != 0:
      effects[e][f] = effects[e][f]/counter[e][f]
    else:
      effects[e][f] = 10000
```

This code computes the means of various subsets of the expression data. Using the numeric extensions to Python, this code becomes

```
effects[e] = where(equal(counter[e], 0), 10000, effects[e])
effects[e] /= where(equal(counter[e], 0), 1, counter[e])
```

In other words, a collection of nested loops affecting element-by-element computations are replaced by matrix-level operations under the control of "mask" matrices, executed by a compiled library function. This transformation is relatively straightforward for someone with a background in software systems for linear algebra, but not at all obvious to the biologist developing the analysis code.

The second major modification involved including low-level "pickling" operations for I/O of Python objects. The program was storing a number of intermediate results on disk and rereading them later in the analysis. Pickling is a term for storing such results in the format of Python's internal object structures, which in turn facilitates the later reading of that data back into Python.

These two changes (the shift to numeric Python and to low-level pickling operations) resulted in an eightfold reduction in run time. After a careful review, the code's developer also restructured and simplified the statistical processing, reducing run time even more, effectively obviating further HPC refinement at least for the present. This case illustrates that once one starts to optimize code with an eye to HPC, one may achieve acceptable performance even without the need for parallel computation.

Current Status and Future Directions

Our work on developing, refining, and applying an HP/LM approach to biomedical HPC is still a work in progress. As of July 2004,

1. The parallelization of Blast (case 1) is now fully operational and being used for various projects.
2. A version of the parallelized microarray analysis program (case 4) is heavily used, and a new cluster has recently been installed to increase the available processing power.

3. The other two projects (cases 2 and 3) are still in the process of iterative algorithmic research. The modest HPC-related algorithmic changes are now part of code. The knowledge that further performance gains can be achieved using parallel computation provides the respective research groups with the opportunity of contemplating larger problems in the near future.

Looking to the future, we are continuing to work with biomedical researchers to extend the use of the HP/LM HPC approach. In addition, we are particularly interested in exploring a number of more fundamental research issues that relate to the approach. These research issues include (1) developing approaches to facilitate code reuse, including dealing with issues of interfaces, data transformations, and nomenclature and (2) performance enhancement of rapid application development environments, for example, the development of optimization techniques and the addition of coordination facilities to facilitate parallel and distributed execution.

Discussion and Lessons Learned

This section discusses a number of the issues that arose in the course of applying the HP/LM approach to HPC in the four case studies described above.

Summary of HP/LM HPC Techniques

In this paper, we have attempted to illustrate a range of HP/LM techniques that can be taken to achieve HPC while still allowing the original program code and the associated bioinformatics project the flexibility to continue to perform algorithmic exploration in a flexible fashion. These techniques include the following.

1. One very simple but powerful approach involves making modest local modifications to the program code that very selectively introduce efficiency, after first profiling that code to identify areas of the code that are highly computationally intensive. Commonly useful techniques in this regard are the extraction of common subexpressions from inside nested loops so that they are only computed once and strength reduction to simplify computations that do need to be done every iteration. There are many other techniques that an HPC expert can bring to bear once an appropriate collaborative relationship has been established.
2. Another useful technique is to use optimized, compiled libraries to accomplish well-defined operations rather than include the code to perform these tasks in the user's code. This approach is particularly useful in allowing the user to continue to work within an interpretive rapid application development (RAD) environment, such as R or Python, while still achieving significant performance enhancements. Examples might include matrix operations, sorting, and specialized algorithms such as singular value decomposition. An important component of our collaborative approach involves helping the bioinformaticist to realize that the particular problem he or she is confronting could profitably use an existing special-purpose library of this sort.
3. Parallel computation is, of course, a centrally important technique for achieving HPC. Taking the HP/LM approach, however, it is often possible to take a user's code and parallelize it in a way that (a) does not involve making many changes to that code, (b) allows the structure of the code to remain unchanged, and (c) allows the code to be refined and tested in a sequential (single processor) mode but run operationally in parallel.
4. Finally, the porting of a program to parallel operation can require confronting a number of somewhat idiosyncratic, and perhaps unexpected, issues. Examples discussed above include (a) the need for specialized error checking (in the case of Blast), (b) the need to coordinate the use of random number generators so that the parallel program will produce the identical output as the sequential version of a program, and (c) in some circumstances, restructuring and rationalizing the program code to make it easier to work in either the sequential or parallel mode.

As mentioned previously, we used the parallel programming environment, Linda, in cases 2 and 3. Although the use of Linda is in no way critical to the HP/LM approach, the use of a coordination facility that supports asynchronous access to a shared variable greatly simplifies the realization of a dynamic, adaptive partitioning of the work in these two cases. This style of coordination can in principle be achieved using other approaches to parallel programming such as message passing systems like PVM[5] or MPI,[6] but not nearly as easily. Gropp et al.,[6] for example, present an MPI example of a shared counter that runs to three pages of code. Carriero and Gelernter[24] discuss this topic in more detail.

Treating the Bioinformatics Program Code as an Executable Specification

When working with actual code, arguably the most important lesson is that, in this HPC process, one should approach the code as an executable specification. In other words, the code is often the most precise statement available in any form of the analyses/computations of interest. It is a snapshot that captures one moment in a research group's ongoing effort to tackle a difficult problem. As that research group's understanding evolves, the new understanding is embodied in changes to the code. We want to impose few, if any, constraints on this essentially creative process.

The main implication of this is that code changes made with an eye to performance improvement are nonetheless changes to this specification and so should be made in a manner that has minimal impact on that specification. Generally, this is easier to accomplish with small rather than large changes, but even the latter are possible. Indeed, when the latter are appropriate, they can have the beneficial effect of significantly simplifying the specification. For example, replacing in-line linear algebra operations with calls to library routines makes it clearer what the code is actually doing and will often run significantly faster.

How Well Will a "Minimalist" Approach Work?

In the broad context of computer science research, the HP/LM HPC approach does not involve high-tech innovation in the extreme "big iron" sense, although there is certainly a wealth of interesting and challenging research questions in adapting the approach to the many needs of biomedicine. In a sense, the HP/LM approach is an attempt to apply the 90-10 rule—that a substantially large portion (approximately 90%) of a potential gain can often be realized with a relatively small portion (approximately 10%) of the effort needed to achieve that whole gain. This paper describes four case studies that illustrate this philosophy and attempts to distill some basic lessons from this experience. The most important result is the positive finding that in all cases, the 90-10 rule held and respectable performance gains were possible with modest effort and little impact on the base software systems.

One of the projects involved a base system (Blast) that was treated as atomic. For Blast, no internal changes were made, rather it was augmented with external components to improve performance. Three projects involved base codes that were modified in modest ways. Although it is hard to quantify effort in detailed terms, we can give a general sense by noting that for the primary computer science researcher who performed this work, approximately one half staff-year was invested directly in the four projects described. Further, for none of the bioinformaticists did the HPC process become an end in itself, rather it complemented ongoing development efforts of the base "specification." Significant reductions in run time were achieved across the board as a result of this modest investment of effort. In addition, the process often led to a sounder code base.

An HP/LM Model for Biomedical HPC Collaboration

Implicit in the HP/LM approach to HPC described in this paper is a corresponding HP/LM model of biomedical HPC collaborations. As the complexity of bioinformatics analyses increases and the need for HPC is increasingly encountered, the question arises as to how best to structure collaborations that address these problems. For example, (1) bioinformaticists might become steeped in the lore of HPC, (2) computer scientists might invest the time and energy to become knowledgeable bioinformaticists, and (3) large, highly computational projects might serve as strong bonds to unite bioinformatics and computer scientists in joint collaborative projects.

The first two solutions are potentially limited given the fact that few individuals are likely to be able to master a significant fraction of both fields. The third approach seems logical, but it is often realized as a tight collaboration of bioinformaticists and computer scientists over an extended period of time, a model that is so resource intensive that wide deployment is inhibited. The third approach, however, does suggest an HP/LM model of collaboration based on more loosely bound, less resource-intensive interactions. In summary, (1) accept that the goal is a performance level that shifts the rate-limiting step elsewhere, not the achievement of some provably optimal run time; (2) respect the base code/system; (3) focus on the minimal set of changes needed to realize the performance goal.

This general approach will in turn (1) reduce cost and time of developing high-performance bioinformatics application solutions, (2) tend to insulate research and operational application software from system specifics, and (3) reduce the cost of maintaining and enhancing applications for both the code developers and HPC specialists.

These principles will steer the collaboration in a direction that could require relatively little intervention on the part of the computer scientists and relatively little adjustment on the part of the bioinformaticists. In this way, the performance goal can be met and expertise exchanged without the need for creating an extensive framework for collaboration. If this approach is tenable, then the door is opened to much broader deployment of HPC in bioinformatics in that smaller projects can be considered that would not otherwise have justified a more substantial effort.

References ■

1. Arlauskas R. iPSC/2 system: a second generation hypercube. In: Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues. New York: ACM Press, 1988:32–42.
2. Johnsson SL. The connection machine systems CM-5. Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures. New York: ACM Press, 1993:365–6.
3. ASCI Red, 1998. Available at: http://www.sandia.gov/ASCI/Red/. Accessed Oct 2004.
4. Blue Gene, 2002. Available at: http://www.research.ibm.com/bluegene/. Accessed Oct 2004.
5. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek M, Sunderam S. PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing. Cambridge, MA: MIT Press, 1984.
6. Gropp W, Lusk E, Skjellum A. Using MPI-2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). Cambridge, MA: MIT Press, 1999.
7. Bjornson R, Carriero N, Gelernter D. From weaving threads to untangling the web: A view of coordination from Linda's perspective. In: Garlan D, LeMétayer D (eds). Coordination Languages and Models. New York: Springer, 1997:1–17.
8. Dwarkadas S, Schäffer AA, Cottingham RW, Cox AL, Keleher P, Zwaenepoel W. Parallelization of general linkage analysis problems. Hum Hered. 1994;44:127–41.
9. Turboblast. Available at: http://www.turboworx.com/solutions/turboblast/. Accessed Oct 2004.
10. Janaki C, Joshi R. Accelerating comparative genomics using parallel computing. In Silico Biology. 2003;3(4):429–40.
11. Li X, Gu W, Mohan S, Baylink DJ. DNA microarrays: their use and misuse. Microcirculation. 2002;9:13–22.
12. Forster T, Roy D, Ghazal P. Experiments using microarray technology: limitations and standard operating procedures. Endocrinology. 2003;178:195–204.
13. Templin MF, Stoll D, Schrenk M, Traub PC, Vohringer CF, Joos TO. Protein microarray technology. Trends Biotechnol. 2002;20:160–6.
14. Patton WF, Schulenberg B, Steinberg TH. Two-dimensional gel electrophoresis; better than a poke in the ICAT? Curr Opin Biotechnol. 2002;13:321–8.
15. Bentley J. Programming Pearls. 2nd ed. Reading, MA: Addison-Wesley, 2000.
16. Kernighan B, Pike R. The Practice of Programming. Reading, MA: Addison-Wesley, 1999.
17. Aho AV, Sethi R, Ullman JP. Compilers: Principles, Techniques and Tools. Reading, MA: Addison-Wesley, 1986.
18. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Mol Biol. 1990;215:403–10.
19. Harrison PM, Carriero NJ, Liu Y, Gerstein M. A PolyORFomic ANalysis of prokaryote genomes using disabled-homology filtering reveals conserved but undiscovered short ORFs. J Mol Biol. 2003;333:885–92.
20. Fung ET, Wright GL, Dalmasso EA. Proteomic strategies for biomarker identification: progress and challenges. Curr Opin Mol Ther. 2000;2:643–50.
21. Petricoin EF, Ardekani AM, Hitt BA, et al. Use of proteomic patterns in serum to identify ovarian cancer. Lancet 2002;359:572–7.
22. Breiman L. RandomForest, Technical Report. Berkeley, CA: Statistics Department, University of California, 2001.
23. Wu B, Abbott T, Fishman D, et al. Comparison of statistical methods for classification of ovarian cancer using a proteomics dataset. Bioinformatics. 2003;19:1636–43.

24. Carriero NJ, Gelernter DH. Some simple and practical strategies for parallelism. Algorithms for parallel processing. In: Heath M, Randade A, Schreiber R (eds). IMA Volumes in Mathematics and Applications, Volume 105. New York: Springer-Verlag, 1998: 75–88.
25. Carriero N, Gelernter D. Linda in context. Communications ACM. 1989;32:444–58.
26. Zhang HP, Feng R, Zhu HT. A latent variable model of segregation analysis for ordinal traits. J Am Stat Assoc. 2003;98:1023–34.
27. Yang YH, Dudoit S, Lu P, Speed TP. Normalization for cDNA Microarray Data. SPIE BiOS 2001, San Jose, CA, 2001.
28. Kerr MK, Martin M, Churchill GA. Analysis of variance for gene expression microarray data. J Comput Biol. 2000;7:819–37.
29. Rifkin S, Kim J, White K. Evolution of gene expression in the Drosophila melanogaster subgroup. Nat Genet. 2003;33:138–44.
30. Kerr MK, Churchill GA. Statistical design and the analysis of gene expression microarrays. Genet Res. 2001;77:123–8.
31. Wu H, Kerr MK, Cui X, Churchill GA. MAANOVA: A software package for the analysis of spotted cDNA microarray experiments. In: Parmigiani G, Garrett ES, Irizarry RA, Zeger SL (eds). The Analysis of Gene Expression Data: Methods and Software. New York: Springer, 2003.