Scientific Programming 16 (2008) 277–285
DOI 10.3233/SPR-2008-0260
IOS Press

Multicore challenges and benefits for high performance scientific computing

Ida M.B. Nielsen [a] and Curtis L. Janssen [a,*]
[a] *Sandia National Laboratories, P.O. Box 969, Livermore, CA 94551, USA*

Abstract. Until recently, performance gains in processors were achieved largely by improvements in clock speeds and instruction level parallelism. Thus, applications could obtain performance increases with relatively minor changes by upgrading to the latest generation of computing hardware. Currently, however, processor performance improvements are realized by using multicore technology and hardware support for multiple threads within each core, and taking full advantage of this technology to improve the performance of applications requires exposure of extreme levels of software parallelism. We will here discuss the architecture of parallel computers constructed from many multicore chips as well as techniques for managing the complexity of programming such computers, including the hybrid message-passing/multi-threading programming model. We will illustrate these ideas with a hybrid distributed memory matrix multiply and a quantum chemistry algorithm for energy computation using Møller–Plesset perturbation theory.

Keywords: Multicore technology, hybrid programming model, multi-threading, message-passing, matrix multiplay

1. Introduction

On-chip performance gains, which have previously been realized by increases in instruction level parallelism and clock speed, are now obtained by using multicore technology, that is, by placing multiple, logically independent, processing cores on each chip. This is illustrated in Fig. 1, which depicts the development over the past couple of decades of the clock rate and relative floating point performance for the fastest available single chips. The relative floating point performance was computed by dividing the floating point performance by the clock rate and normalizing to make the relative performance equal to unity for the first data point. It is apparent from the figure that the growth in clock rates has leveled off (and even reversed); also, the relative single-chip performance has increased dramatically over the past few years, and this sudden increase in relative performance has been achieved by using multicore chips. Improvements in instruction level parallelism will continue to be achieved by giving each core the ability to execute several threads of execution, a technique known as simultaneous multi-threading (SMT). This allows a core to process one thread while another thread is waiting, e.g., for data to be loaded from memory. These trends indicate that to take full advantage of even a single processing chip, it will be necessary, in the near future, for programmers to expose a high level of parallelism within their applications.

The performance gains displayed for the fastest parallel computers in recent years are, in part, attributable to the use of multicore chips. Figure 2 depicts the development in the (absolute) single-chip floating point performance and the performance of the fastest parallel supercomputers, the latter of which is represented by the TOP500 HPLINPACK benchmark [1] for the top machine. Supercomputer performance has increased significantly faster than the single-chip performance, and this faster growth rate has been achieved by continually increasing the number of processing chips employed. The combination of multicore technology and SMT dramatically increases the levels of parallelism that must be exposed in applications to achieve good performance on large-scale parallel computers in the near future. Programs designed for today's parallel computers will be unlikely to obtain good performance on machines available in a few years' time.

In the following we will first briefly discuss the salient features of the hybrid shared-distributed memory architectures provided by parallel computers that are constructed from multicore chips. We will then discuss strategies for efficiently utilizing such archi-

*Corresponding author. E-mail: cljanss@sandia.gov.

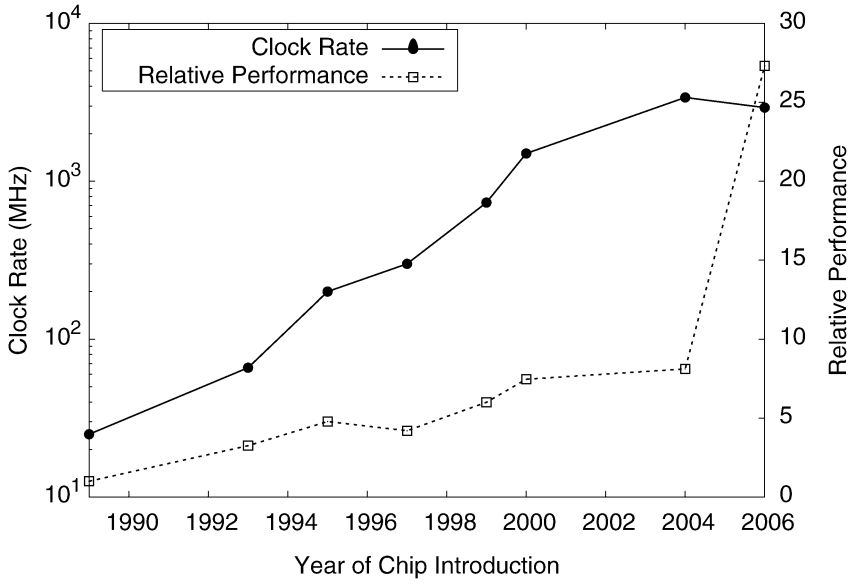

Fig. 1. Clock rate (MHz) and floating point performance (relative to the clock rate) trends for the fastest single chip, at time of introduction. Floating point results were normalized to make the relative floating point performance in 1989 equal to unity.

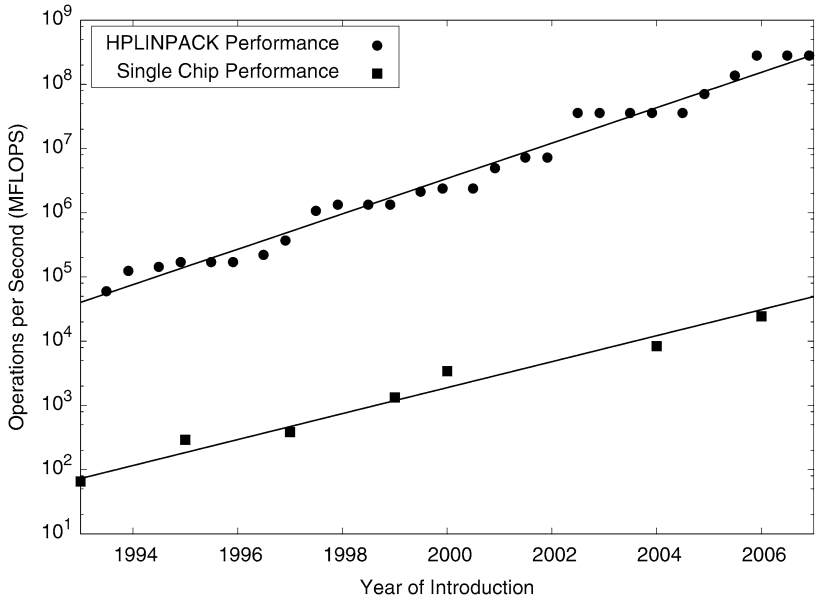

Fig. 2. Floating point performance of a single chip compared to the floating point performance of the fastest parallel machine, the latter represented by the HPLINPACK result from the top machine in the TOP500 benchmark [1]. The single chip performance doubles roughly once every 1.5 years and the speed of the fastest parallel machine doubles approximately every year.

tectures using programming models that employ both multi-threading and message-passing. Specific examples of parallel hybrid algorithms will be given, including a matrix-matrix multiplication and a quantum chemistry method, namely second-order Møller–Plesset perturbation theory [2].

2. Hybrid shared-distributed memory architecture

The use of multicore technology in parallel computers creates systems with multi-layered memory hierarchies that pose special challenges for development of efficient parallel software. In Fig. 3 we show the

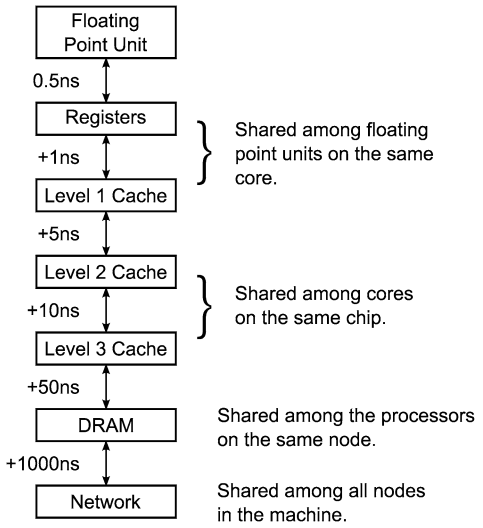

Fig. 3. An illustration of the deep memory hierarchy in a parallel computer. The top box represents the ultimate consumer and producer of data (often a floating point unit), and the other boxes represent different stages in the memory hierarchy. The times listed between boxes are the incremental times needed to obtain data from the next level in the hierarchy.

various stages in the memory hierarchy of a parallel computer whose nodes consist of multiple chips that each contain multiple cores. At the most basic level in this hierarchy, data is stored in registers that are allocated to application variables by the compiler. Several layers of memory cache are typically provided to hide the relatively slow access time to the system's main memory. Each layer within the memory cache hierarchy becomes larger and slower as the data moves further away from the core. The system memory typically employs Dynamic Random Access Memory (DRAM) technology, which is substantially slower than the memory cache. Finally, if data is required that is not stored on the local node, then a high speed network must be utilized to access that data, at a yet higher latency cost. Because multiple cores are involved in simultaneous memory access throughout the memory hierarchy, bottlenecks can arise. When multiple cores temporarily require exclusive access to the same memory location, locking protocols must be employed that introduce yet more latency into the system. An additional complication, not illustrated in Figure 3, is the possible presence of non-identical processing cores in the system, each with their own memory hierarchy. The consequence of the complex structure of modern computing systems is that programmers, and the tools they use, must structure the program and the generated object code to reuse local data to the greatest extent possible, all while exposing extreme levels of parallelism.

3. Managing the complexity of programming hybrid memory architectures

The rapid increase in the number of cores employed in large-scale parallel computers creates a need for developing software capable of utilizing extreme levels of parallelism. Reexamination of current programming practices is required to expose additional levels of parallelism and to provide a means for taking advantage of the hybrid shared-distributed memory architectures provided by machines constructed from multicore chips. A two-pronged approach will be required to take full advantage of future architectures: Viable strategies are needed for retrofitting existing parallel applications in which significant investments have already been made, and new programming approaches must be developed to facilitate development of new high-performance low-maintenance applications for future computers.

A commonly used strategy for programming applications to run on a hybrid shared-distributed memory architecture is a so-called hybrid parallelization approach that employs both multi-threading and message-passing. This approach is more complex than using either multi-threading or message-passing on its own, but it offers the potential for improved performance by employing multi-threading, rather than multiple processes, for parallelization within a node. Thus, the memory accessed in a read-only mode by multiple threads on a node can be shared, reducing the demands for system memory, and synchronization and communication between different threads in the same process is very fast, allowing finer degrees of parallelism to be exposed. Because of the accelerating rate of increase in the total number of cores in parallel computers, it will be vital to expose as much parallelism as possible to utilize future machines. Although new programming models are being developed to help manage the complexity of programming large-scale machines, hybrid message-passing and multi-threaded programming techniques are currently the most viable parallel programming model available.

While the sharing of memory between the threads within a process is advantageous in some cases, it also makes multi-threaded programming more difficult and complex. Most programmers of high performance scientific applications are accustomed to the complete and

detailed control provided when data is shared by explicit communication as provided by message-passing libraries, but the memory sharing between threads makes it possible for one thread to inadvertently modify the memory that another thread is using. Care must therefore be exercised to avoid this situation, and a combination of mutual exclusion locks and the judicious use of data replication can be used to successfully use a multi-threaded programming environment.

Below we will discuss a couple of examples of the use of a hybrid multi-threaded/message-passing programming model for scientific computing applications, beginning with the simple parallel distributed memory hybrid matrix multiply algorithm covered in Section 3.1. As mentioned above, retrofitting existing code to take advantage of a hybrid memory architecture may be necessary to improve its performance, and in Section 3.2 we will discuss how an object-oriented programming technique was used to manage the complexity of retrofitting a parallel quantum chemistry algorithm to use a hybrid programming model.

3.1. Hybrid parallel matrix–matrix multiplication

We will consider a parallel matrix–matrix multiplication algorithm based on an existing algorithm [3] that employs message-passing for parallelization across nodes and multi-threading for intra-node parallelism. The algorithm computes the product $\mathbf{C} = \mathbf{AB}$, and to simplify the discussion we will assume that all matrices are of dimension $n \times n$. Additionally, we will require that the number of nodes, p, is a square number and that $\sqrt{p}$ is a divisor of n.

All matrices are distributed by blocks across nodes, and there are p blocks, which are all of dimension $(n/\sqrt{p}) \times (n/\sqrt{p})$. The nodes are labeled as P_{ij}, where i and j both run from 0 through $\sqrt{p} - 1$, and node P_{ij} holds the block $\mathbf{C}_{ij}$ of $\mathbf{C}$ throughout the computation. Additionally, at any given time, P_{ij} will store the blocks $\mathbf{A}_{il}$ and $\mathbf{B}_{lj}$ of $\mathbf{A}$ and $\mathbf{B}$ for one value of l ($0 \leqslant l < \sqrt{p}$), which varies as the computation progresses. The computation of the $\mathbf{C}_{ij}$ block can be written as a sum over l of products of blocks $\mathbf{A}_{il}$ and $\mathbf{B}_{lj}$

$$\mathbf{C}_{ij} = \sum_{l=0}^{\sqrt{p}-1} \mathbf{A}_{il}\mathbf{B}_{lj} \tag{1}$$

and work is distributed across nodes by letting each node P_{ij} compute the corresponding block, $\mathbf{C}_{ij}$, of $\mathbf{C}$. The block $\mathbf{C}_{ij}$ contains contributions from block pairs $\mathbf{A}_{il}$ and $\mathbf{B}_{lj}$ for all l, and all of these block pairs must therefore visit P_{ij} during the computation. This is accomplished by means of a systolic loop communication pattern, which works as follows. The nodes are positioned on a logical $\sqrt{p} \times \sqrt{p}$ grid with nodes P_{il}, $0 \leqslant l < \sqrt{p}$, in row i and nodes P_{lj}, $0 \leqslant l < \sqrt{p}$, in column j. During execution, the blocks $\mathbf{A}_{il}$ will be sent around in a ring pattern within row i, and the $\mathbf{B}_{lj}$ blocks, analogously, will be sent around within column j; wraparound is used at the end of the rows and columns to generate a closed ring. The communication is set up so that blocks $\mathbf{A}_{il}$ and $\mathbf{B}_{lj}$ with a common value for l always visit a node simultaneously, as required to compute a contribution to $\mathbf{C}_{ij}$ (Eq. (1)). To achieve this, we use the initial distribution of the $\mathbf{A}$ and $\mathbf{B}$ matrices shown in Fig. 4; during execution, blocks of $\mathbf{A}$ will be sent to the right within a row and blocks of $\mathbf{B}$ will be sent down within a column, with $P_{i,\sqrt{p}-1}$ sending blocks of $\mathbf{A}$ to $P_{i,0}$ and $P_{\sqrt{p}-1,j}$ sending blocks of $\mathbf{B}$ to $P_{0,j}$ to close the loops.

The systolic loop employed in the algorithm is shown in Fig. 5. The number of steps required to complete the multiplication equals $\sqrt{p}$, where a step is defined as the computation on a node of the contribution to the local block of $\mathbf{C}$ from the currently local blocks of $\mathbf{A}$ and $\mathbf{B}$ followed by the transmission of $\mathbf{A}$ and $\mathbf{B}$ blocks between neighboring nodes; note that the last step does not require communication because the local $\mathbf{A}$ and $\mathbf{B}$ blocks are not needed afterwards. The algorithm employs nonblocking sends and receives (MPI_Isend and MPI_Irecv); the use of nonblocking communication prevents deadlock in the systolic loop and also allows communication to be overlapped with computation for increased efficiency. The degree to which computation is actually overlapped with communication depends strongly on the MPI implementation and the particular network employed. If a rendezvous protocol is used, where the actual send of the data does not begin until the receiving side explicitly indicates that it is ready for the transfer, then it is possible for the communication to not begin until MPI_Waitall is called after the computation of the local matrix multiply, thus preventing the overlap of communication and computation. This can be prevented by using an MPI implementation that can make progress on communication even when the process is otherwise engaged in computation. For the MPI implementation used in this example, such progress is not made by default. To enable overlap, we create a separate thread that MPI can use to make progress, even when all of the application's other threads are engaged exclusively in computation [4].

	$j=0$	$j=1$	$j=2$	$\cdots$	$j=\sqrt{p}-1$
$i=0$	$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{1,1}$	$A_{0,2}$ $B_{2,2}$	$\cdots$	$A_{0,\sqrt{p}-1}$ $B_{\sqrt{p}-1,\sqrt{p}-1}$
$i=1$	$A_{1,1}$ $B_{1,0}$	$A_{1,2}$ $B_{2,1}$	$A_{1,3}$ $B_{3,2}$	$\cdots$	$A_{1,0}$ $B_{0,\sqrt{p}-1}$
$i=2$	$A_{2,2}$ $B_{2,0}$	$A_{2,3}$ $B_{3,1}$	$A_{2,4}$ $B_{4,2}$	$\cdots$	$A_{2,1}$ $B_{1,\sqrt{p}-1}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$i=\sqrt{p}-1$	$A_{\sqrt{p}-1,\sqrt{p}-1}$ $B_{\sqrt{p}-1,0}$	$A_{\sqrt{p}-1,0}$ $B_{0,1}$	$A_{\sqrt{p}-1,1}$ $B_{1,2}$	$\cdots$	$A_{\sqrt{p}-1,\sqrt{p}-2}$ $B_{\sqrt{p}-2,\sqrt{p}-1}$

Fig. 4. Initial distribution of the **A** and **B** matrices across a $\sqrt{p} \times \sqrt{p}$ grid of nodes: the node $P_{i,j}$ holds the blocks $\mathbf{A}_{i,(i+j)(\mathrm{mod}\ \sqrt{p})}$ and $\mathbf{B}_{(i+j)(\mathrm{mod}\ \sqrt{p}),j}$.

```
/* Initialize matrices ... */

/* Begin systolic loop */
for (int index=0; index<sqrt_nproc-1; index++) {
    /* Begin receiving the next A_local block into a temporary array. */
    MPI_Irecv(tmp_A_local[0], ndata, MPI_DOUBLE,
              prev_proc_in_row, 0, comm, &reqs[0]);
    /* Begin receiving the next B_local block into a temporary array. */
    MPI_Irecv(tmp_B_local[0], ndata, MPI_DOUBLE,
              prev_proc_in_col, 0, comm, &reqs[1]);
    /* Begin sending the A_local block to the next proc in our row. */
    MPI_Isend(A_local[0], ndata, MPI_DOUBLE,
              next_proc_in_row, 0, comm, &reqs[2]);
    /* Begin sending the B_local block to the next proc in our column. */
    MPI_Isend(B_local[0], ndata, MPI_DOUBLE,
              next_proc_in_col, 0, comm, &reqs[3]);

    /* Compute contribution to C_local from A_local and B_local */
    mxm(C_local, A_local, B_local, local_n);

    /* Swap the local matrix pointers with those for the temporary
       arrays used to receive the next local block. */
    swap_arrays(&A_local,&tmp_A_local);
    swap_arrays(&B_local,&tmp_B_local);

    /* Wait for data to finish being sent or received before proceeding */
    MPI_Waitall(4,reqs,stats);
  }

/* Compute the final contribution to C_local from A_local and B_local */
mxm(C_local, A_local, B_local, local_n);
```

Fig. 5. The systolic loop employed in the hybrid parallel matrix–matrix multiplication algorithm.

Intra-node parallelization is achieved by using multiple threads for computation of the local $\mathbf{C}_{ij}$ block as shown in Fig. 6. Multi-threading is here implemented via OpenMP [5] using the pragma directive "parallel for", which causes the following `for` loop to be performed in parallel by starting up a number of threads (determined at runtime) on each node. The computation of $\mathbf{C}_{ij}$ is parallelized over i, each thread being responsible for its own subset of i values, so that different threads will not attempt to update the same block $\mathbf{C}_{ij}$ simultaneously.

The parallel performance of the hybrid matrix-matrix multiplication algorithm was investigated using a Linux cluster [6]. Four sets of runs were performed, employing one or two computation threads per node, and in each case running both with and without a communication progress thread. All cases were performed with a problem size of $n = 5040$ running on a number of nodes ranging from 1 to 36. We first note that, running on a single node, the speedup when increasing the number of threads from one to two is 1.7, corresponding to an efficiency of 86%. Because the two processors must compete for shared system resources

```
void mxm(double **C_local, double **A_local, double **B_local,
         int local_n)
{
#pragma omp parallel for
  for (int i=0; i<local_n; i++) {
    for (int j=0; j<local_n; j++) {
        C_local[i][j] += dot(&A_local[0][i*local_n],
                             &B_local[0][j*local_n], local_n);
    }
  }
}
```

Fig. 6. Multi-threaded part of the hybrid matrix–matrix multiplication, multiplying the local blocks of **A** and **B** and adding the result into the local block of **C**.

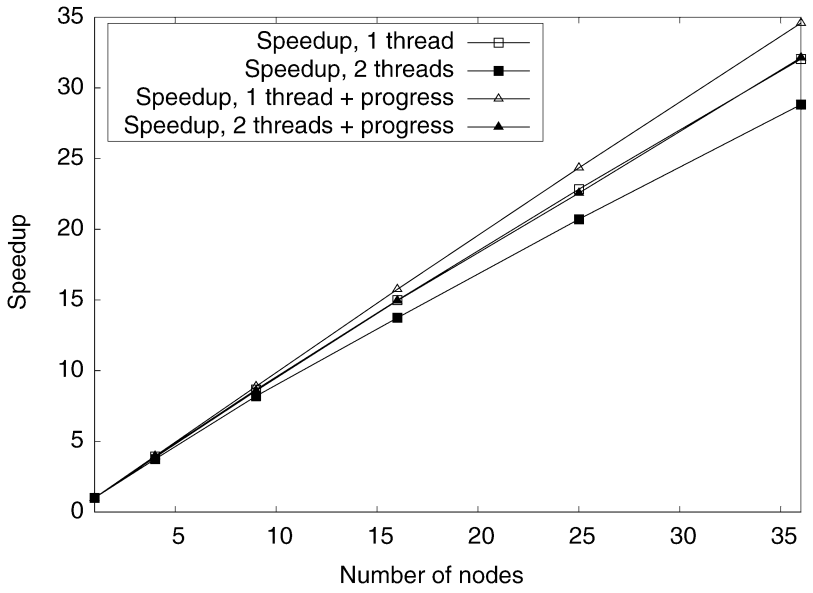

Fig. 7. Parallel speedups for the hybrid matrix–matrix multiplication using one or two computation threads with and without a progress thread.

and implement cache coherency, perfect speedups are typically not obtained when using multiple threads on a node.

Figure 7 illustrates the speedups, computed relative to the number of nodes (not threads), obtained for the four sets of runs. When running without a progress thread, the performance is somewhat less than ideal with speedups of 32 and 29 obtained on 36 nodes using one and two computation threads per node, respectively, corresponding to efficiencies of 89% and 80%. The amount of computation performed per node is $O(n^3/p)$, while the amount of communication per node is $O(n^2/\sqrt{p})$, and, hence, the communication to computation ratio, $O(\sqrt{p}/n)$, increases with the number of nodes for a fixed problem size, and the algorithm is not strongly scalable. As mentioned above, however, the employed communication scheme in the systolic loop, which uses nonblocking send and receive operations, is designed to allow overlap of communication and computation, prefetching new blocks of the **A** and **B** matrices while computing the product of the current blocks. When a progress thread is used to permit this overlap, the communication overhead is hidden in the systolic loop, and the measured communication time is negligible in all cases. In this case, speedups on 36 nodes are 35 and 32 when using one and two computation threads per node, respectively, corresponding to efficiencies of 96% and 89% – a substantial improvement over the runs without a progress thread. Initially, however, blocks of **A** and **B** must be redistributed to create the distribution shown in Fig. 4; this communication step cannot be overlapped with computation, and it is the cause for the slightly less than ideal performance observed.

3.2. A parallel hybrid quantum chemistry algorithm

As an example of a hybrid programming approach in a complex scientific application, we will discuss a

quantum chemistry algorithm, which computes energies with second-order Møller–Plesset (MP2) perturbation theory. Providing a good compromise between high accuracy and low computational cost, the MP2 method is a widely used quantum chemical method for computing molecular energies and properties when higher accuracy is sought than offered by the basic Hartree–Fock method.

The MP2 energy is given as the sum of the Hartree–Fock energy and the MP2 correlation energy, and the latter is computed from the two-electron integrals in the molecular orbital basis, $(ia|jb)$, as follows

$$E_{\mathrm{MP2}}^{\mathrm{corr}} = \sum_{ijab} \frac{(ia|jb)[2(ia|jb) - (ib|ja)]}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b} \tag{2}$$

where i and j represent occupied molecular orbitals, a and b denote unoccupied molecular orbitals, and ϵ_m designates a molecular orbital energy. The two-electron integrals in the molecular orbital basis are generated from the atomic orbital integrals, $(\mu\nu|\rho\sigma)$, by a four-index transformation

$$(ia|jb) = \sum_{\mu\nu\rho\sigma} C_{\mu i} C_{\nu a} C_{\rho j} C_{\sigma b} (\mu\nu|\rho\sigma) \tag{3}$$

where the transformation coefficients, $C_{\mu i}$ etc., are the molecular orbital coefficients computed in the Hartree–Fock procedure, and μ, ν, ρ, and σ represent atomic orbitals.

This integral transformation is the most time consuming part of the MP2 computation, and it is usually performed as four separate quarter transformations. In the computation of the atomic orbital integrals, related atomic orbitals are grouped into so-called shells for computational efficiency, and the four quarter transformations can be then be written as

$$(iN|RS) = \sum_{M} C_{Mi} (MN|RS), \tag{4}$$

$$(iN|jS) = \sum_{R} C_{Rj} (iN|RS), \tag{5}$$

$$(ia|jS) = \sum_{N} C_{Na} (iN|jS), \tag{6}$$

$$(ia|jb) = \sum_{S} C_{Sb} (ia|jS), \tag{7}$$

where M, N, R, and S are shells of atomic orbitals with $\mu \in M$, $\nu \in N$, $\rho \in R$, and $\sigma \in S$, and the quantities $(MN|RS)$, $(iN|RS)$, $(iN|jS)$, and $(ia|jS)$ each refer to a block of integrals.

Efficient parallelization of the integral transformation is the key to a high-performance parallel MP2 program. Because the number of two-electron integrals can be very large (there can be many billions of integrals for a medium-sized molecule), the two-electron integrals must be distributed, and this necessitates communication during the integral transformation. Moreover, because the computational tasks are of very nonuniform size, one-sided communication must be employed to minimize the time processes spend waiting for required data to arrive.

In Fig. 8 we show a parallel algorithm [7] for performing the integral transformation using a hybrid message-passing and multi-threading approach [8], which is implemented in the Massively Parallel Quantum Chemistry program [9]. In this algorithm, every node runs both computation and communication threads. The bulk of the floating point intensive work is carried out by the computation threads, which compute the atomic orbital integrals and transform them into the molecular orbital basis. The communication (exchange of integrals between processes) required during the integral transformation is handled transparently by the

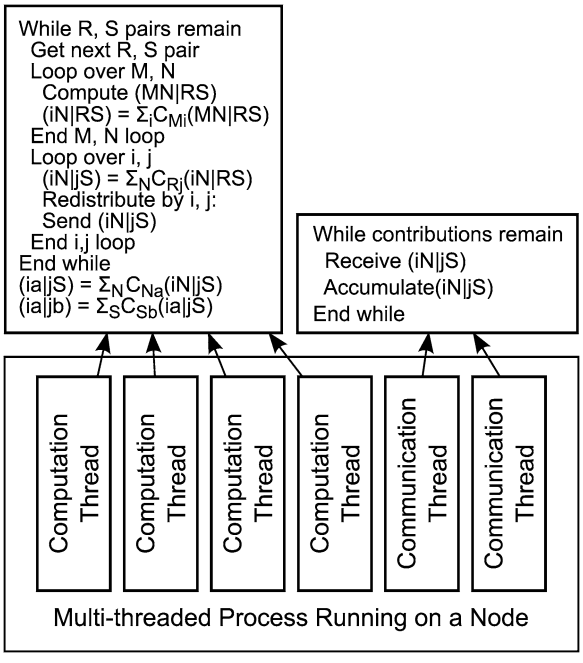

Fig. 8. A schematic illustrating one node of a parallel computer running the hybrid MP2 algorithm. The code performing the bulk of the computation runs in several threads, and the parts of the program performing primarily communication run in one or more separate communication threads.

communication threads, with minimal interruption of the computation on the computation threads. In the first two quarter transformations, work and data are distributed by distribution of R, S shell pairs, and, hence, each compute thread can compute only a partial contribution to the half-transformed integrals $(iN|jS)$. In the third and fourth quarter transformations, a different distribution of work and data, namely distribution over ij pairs, is employed, however, so when a compute thread has computed the contribution to a batch of half-transformed integrals, it sends them to the node that is to store and process these integrals for the remainder of the integral transformation. On the receiving end, a communication thread receives the contributions to the half-transformed integrals and sums them into the appropriate memory locations. Using dedicated communication threads in the redistribution of the half-transformed integrals ensures prompt processing of the received integrals and eliminates the need to explicitly synchronize tasks running on different nodes. After this redistribution of the half-transformed integrals, each compute thread can finish the integral transformation for its own subset of integrals, using only locally stored data.

Finally, let us briefly discuss how the code, which was originally designed to use only message-passing, was retrofitted by means of an object-oriented programming technique to use a combination of multi-threading and message-passing. The algorithm requires the evaluation of two-electron integrals in multiple threads. Each two-electron integral depends on data that is independent of the integral being evaluated as well as intermediate data that is specific to the desired integral. The integral package was originally written in the C programming language, and, to avoid the overhead of repeated allocation of scratch data for each integral evaluation, some amount of scratch data is kept in static variables (that would be shared among all threads). The integral-independent data is also computed and stored in these static variables. This programming pattern is common for many computations in quantum chemistry applications. It was surprisingly straightforward to migrate this pattern into a factory pattern implemented in the C++ programming language. This separates the integral package interface into a portion that is thread-aware and a portion that is not. The integral factory, when constructed, will generate the integral-independent data that can be shared among all threads. It is then used to construct an integral evaluator for each thread, which will evaluate integrals (the computation threads in Fig. 8). The factory gives each evaluator a reference to the shared data, and each evaluator initializes any scratch data that is needed. Every thread can run its own evaluator completely independently of the others without synchronization. This approach creates a clean design that allows the application to reduce its memory footprint without sacrificing performance.

4. Conclusion

A shift in computer architecture towards multicore processing chips will fundamentally change how programmers utilize large-scale parallel machines. Undoubtedly, new programming models will be developed to make it easier for programmers to utilize extreme levels of parallelism, but, in the interim, a hybrid message-passing/multi-threading approach provides an evolutionary method for efficiently utilizing distributed memory parallel machines comprising shared-memory multicore processors. We have illustrated programming with this hybrid approach using two very different cases. In the first case, a simple distributed memory parallel matrix multiply was converted to the hybrid approach by the addition of a single compiler directive. To obtain the best performance in this case it was necessary to provide one additional thread that is used by the MPI library to overlap communication with computation. In principle, it is not necessary for the programmer to explicitly provide this thread – it could be entirely hidden by the MPI library, but, at present, not all MPI implementations support this feature. In the second case, a parallel implementation of second order Møller–Plesset perturbation theory (MP2) was reformulated to more thoroughly use multi-threading, with different threads specialized for different tasks. In practice, effective use of future parallel machines will involve a combination of these approaches.

Acknowledgements

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] H. Meuer, E. Strohmaier, J. Dongarra and H.D. Simon, TOP500 Supercomputing Sites, June 1993–June 2007, http://www.top500.org.

[2] C. Møller and M.S. Plesset, Note on an approximation treatment for many-electron systems, *Phys. Rev.* **46** (1934), 618–622.

[3] R.A. van de Geijn and J. Watts, SUMMA: Scalable universal matrix multiplication algorithm, *Concurrency: Practice and Experience* **9** (1997), 255–274.

[4] S. Sur, H.-W. Jin, L. Chai, and D.K. Panda, RDMA read based rendezvous protocol for MPI over InfiniBand: Design alternatives and benefits, in: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM Press, New York, 2006, pp. 32–39.

[5] OpenMP Application Program Interface, Version 2.5 May 2005, available from http://www.openmp.org.

[6] A Linux® cluster consisting of nodes with two single-core 3.06 GHz Intel® Xeon® processors (each with 512 KiB of L2 cache) connected via a 4X Single Data Rate InfiniBand network with a full fat tree topology.

[7] I.M.B. Nielsen, A new direct MP2 gradient algorithm with implementation on a massively parallel computer, *Chem. Phys. Lett.* **255** (1996), 210–216.

[8] I.M.B. Nielsen and C.L. Janssen, Multi-threading: A new dimension to massively parallel scientific computation, *Comp. Phys. Comm.* **128** (2000), 238–244.

[9] C.L. Janssen, I.B. Nielsen, M.L. Leininger, E.F. Valeev, J.P. Kenny and E.T. Seidl, The Massively Parallel Quantum Chemistry program (MPQC), version 3.0.0-alpha, Sandia National Laboratories, Livermore, CA, USA, 2007, http://www.mpqc.org.