G

GA

►Global Arrays Parallel Programming Toolkit

Gather

►Collective Communication

Gather-to-All

►Allgather

Gaussian Elimination

►Dense Linear System Solvers
►Sparse Direct Methods

GCD Test

►Banerjee's Dependence Test
►Dependences

Gene Networks Reconstruction

►Systems Biology, Network Inference in

Gene Networks Reverse-Engineering

►Systems Biology, Network Inference in

Generalized Meshes and Tori

►Hypercubes and Meshes

Genome Assembly

Ananth Kalyanaraman
Washington State University, Pullman, WA, USA

Synonyms

Genome sequencing

Definition

Genome assembly is the computational process of deciphering the sequence composition of the genetic material (DNA) within the cell of an organism, using numerous short sequences called *reads* derived from different portions of the target DNA as input. The term *genome* is a collective reference to all the DNA molecules in the cell of an organism. *Sequencing* generally refers to the experimental (wetlab) process of determining the sequence composition of biomolecules such as DNA, RNA, and protein. In the context of genome assembly, however, the term is more commonly used to refer to the experimental (wetlab) process of generating reads from the set of chromosomes that constitutes the genome of an organism. Genome *assembly* is the computational step that follows sequencing with the objective of reconstructing the genome from its reads.

Discussion

Introduction

Deoxyribonucleic acid (or DNA) is a double-stranded molecule which forms the genetic basis in most known organisms. The DNA along with other molecules, such as the ribonucleic acid (or RNA) and proteins, collectively constitute the subject of study in various

David Padua (ed.), *Encyclopedia of Parallel Computing*, DOI 10.1007/978-0-387-09766-4,
© Springer Science+Business Media, LLC 2011

branches of biology such as molecular biology, genetics, genomics, proteomics, and systems biology. A DNA molecule is made up of two equal length strands with opposite directionality, with the ends labeled from 5′ to 3′ on one and 3′ to 5′ on the other. Each strand is a sequence of smaller molecules called *nucleotides*, and each nucleotide contains one of the four possible nitrogenous bases – adenine (a), cytosine (c), guanine (g), and thymine (t). Therefore for computational purposes, each strand can be represented in the form of a string over the alphabet $\{a, c, g, t\}$, expressed always in the direction from 5′ to 3′. Furthermore, the base at a given position in one strand is related to the base at the corresponding position in the other strand by the following base-pairing rule (referred to as "base complementarity"): $a \leftrightarrow t, c \leftrightarrow g$. Therefore, the sequence of one strand can be directly deduced from the sequence of the other. For example, if one strand is 5′ *agaccagttac* 3′, then the other is 5′ *gtaactggtct* 3′.

The length of a genome is measured in the number of its base pairs ("bp"). Genomes range in length from being just a few million base pairs in microbes to several billions of base pairs in many eukaryotic organisms. However, all sequencing technologies available till date, since the invention of Sanger sequencing in late 1970s, have been limited to accurately sequencing DNA molecules *no* longer than ~1 Kbp. Consequently, scientists have had to deploy alternative sequencing strategies for extending the reach of technology to genome scale. The most popular strategy is the *whole genome shotgun* (or WGS) strategy, where multiple copies of a single long target genome are shredded randomly into numerous fragments of sequenceable length and the corresponding reads sequenced individually using any standard technology. Another popular albeit more expensive strategy is the hierarchical approach, where a library of smaller molecules called Bacterial Artificial Chromosomes (or BACs) is constructed. Each BAC is ~150 Kbp in length and they collectively provide a minimum tiling path over the entire length of the genome. Subsequently, the BACs are sequenced individually using the shotgun approach.

In both approaches, the information of the relative ordering among the sequenced reads is lost during sequencing either completely or almost completely. Therefore, the primary information that genome assemblers should rely upon is the end-to-end sequence overlap preserved between reads that were sequenced from overlapping regions along the target genome. To increase the chance of overlap, the target genome is typically sequenced in a redundant fashion. This is referred to as *genome coverage*. A higher coverage typically tends to provide information for a more accurate assembly, although at increased costs of generation and analysis. The fact that reads could have originated from an arbitrary strand adds another dimension to the complexity of the reconstruction process. The process is further complicated by other factors such as errors introduced during read sequencing and the presence of genomic repeats that could ambiguate overlap detection and read placement. To partially aid the resolving of the genomic repeat regions, sequencing is sometimes performed in pairs from clonal insert libraries, where the genomic distance between the two reads of a pair can be estimated, typically in the 1–3 Kbp range. This technique is called *pair-end sequencing* and genome assemblers could take advantage of this information to resolve repeats that are smaller than the specified range.

Genome assembly is a classical computational problem in the field of bioinformatics and computational biology. More than two decades of research has yielded a number of approaches and algorithms and yet the problem continues to be actively pursued. This is because of several reasons. While there are different ways of formulating the genome assembly problem, all known formulations are NP-Hard. Therefore, researchers continue to work on developing improved, efficient approximation and heuristic algorithms. However, even the best serial algorithms take tens of thousands of CPU hours for eukaryotic genomes owing to their: large genomic sizes, which increase the number of reads to be analyzed; and high genomic repeat complexity, which adds substantial processing overheads. For instance, in the Celera's WGS version of the human genome assembly published in 2001, over 27 million reads representing 5.11x coverage over the genome were assembled in about $10,000$ CPU h.

A paradigm shift in sequencing technologies has further aggravated the data-intensiveness of the problem and thereby the need for continued algorithmic development. Until the mid-2000s, the only available technology for use in genome assembly projects was Sanger sequencing, which produces reads of approximate length 1 Kbp each. Since then, a slew of

high-throughput sequencing technologies, collectively referred to as the *next-generation sequencing technologies*, have emerged, significantly revitalizing the sequencing community. Examples of next-generation technologies include Roche 454 Life Sciences system's pyrosequencing (read length ~400 bp), Illumina Genome Analyzer and HiSeq (read length ~35–125 bp); Life Technologies SOLiD (read length ~50 bp) and Helicos HeliScope (read length ~35 bp). While these instruments generate much shorter reads than Sanger, they do so at a much faster rate effectively producing several hundred millions of reads in a single experiment, and at significantly reduced costs (about 30–40 times cheaper). These attractive features are essentially democratizing the sequencing process and broadening community contribution to sequenced data. From a genome assembly perspective, a shorter read length could easily deteriorate the assembly quality because the reads are more likely to exhibit false or insufficient overlaps. To offset this shortcoming, sequencing is required at a much higher coverage (30x–200x) than for Sanger sequencing. The higher coverage has another desirable effect in that, because of its built-in redundancy, it could aid in a more reliable identification of real genomic variations and their differentiation from experimental artifacts. Detecting genomic variations such as single nucleotide polymorphisms (SNPs) is of prime importance in comparative and population genomics.

This combination of high coverage and short read lengths makes the short read genome assembly problem significantly more data-intensive than for Sanger reads. For instance, any project aiming to reconstruct a mammalian genome (e.g., human genome) *de novo* using any of the current next-generation technologies would have to contend with finding a way to assemble several billion short reads. This changing landscape in technology and application has led to the development of a new generation of short read assemblers. Although traditional developmental efforts have been targeting serial computers, the increasing data-intensiveness and the increasing complexity of genomes being sequenced have gradually pushed the community toward parallel processing, for what has now become an active branch within the area.

In what follows, an overview of the assembly problem, with its different formulations and algorithmic solutions is presented. This entry is not intended to be a survey of tools for genome assembly. Rather, it will focus on the parallelism in the problem and related efforts in parallel algorithm development. In order to set the stage, the key ideas from the corresponding serial approach will also be presented as necessary. The bulk of the discussion will be on *de novo assembly*, which is typically the harder task of reconstructing a genome from its reads assuming no prior knowledge about the sequence of the genome except for its reads and as available, their pair-end sequencing information.

Algorithmic Formulations of Genome Assembly

The genome assembly problem: Let $R = \{r_1, r_2 \ldots r_m\}$ denote a set of m reads sequenced from an unknown target genome G of length g. The problem of genome assembly is to reconstruct the sequence of genome G from R.

As a caveat, for nearly all input scenarios, the outcome expected is not a single assembled sequence but a *set* of assembled sequences for a couple of reasons. Typically, the genome of a species comprises of multiple chromosomes (e.g., 23 pairs in the human genome) and therefore each chromosome can be treated as an individual sequence target. Note that, however, the information about the source chromosome for a read is lost during a sequencing process such as WGS and it is left for the assembler to detect and sort these reads by chromosomes. Furthermore, any shotgun sequencing procedure tends to leave out "gaps" along the chromosomal DNA during sampling, and therefore it is possible to reconstruct the genome only for those sampled sections. It is expected that, through incorporation of pair-end information, at least a subset of these assembled products (called "contigs" in genome assembly parlance) can be partially ordered and oriented.

Notation and terminology: Let s denote a sequence over a fixed alphabet Σ. Unless otherwise specified, a DNA alphabet is assumed – i.e., $\Sigma = \{a, c, g, t\}$. Let $|s|$ denote the length of s; and $s[i \ldots j]$ denote the substring of s starting and ending at indices i and j respectively, with the convention that string indexing starts at 1. A *prefix i* (alternatively, *suffix i*) of s is $s[1 \ldots i]$ (alternatively, $s[i \ldots |s|]$). Let $n = \Sigma_{i=1}^{m} |r_i|$ and $\ell = \frac{n}{m}$. The sequencing coverage c is given by $\frac{g}{n}$. The terms *string* and *sequence* are used interchangeably. Let p denote the number of processors in a parallel computer.

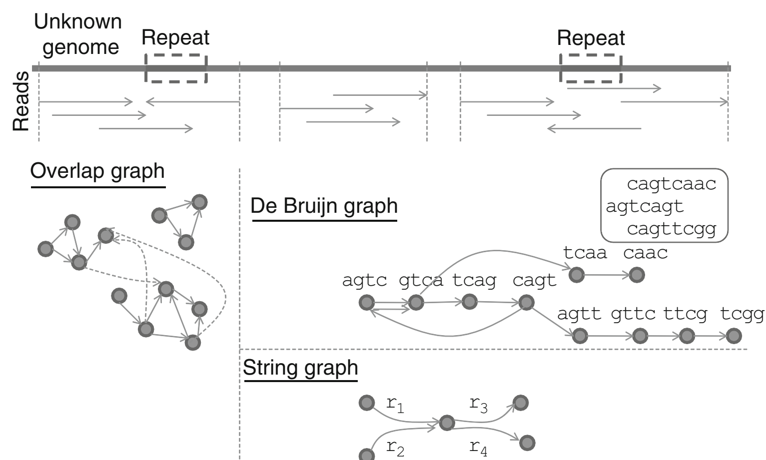
The most simplistic formulation of genome assembly is that of the *Shortest Superstring Problem* (SSP). A *superstring* is a string that contains each input read as a substring. The SSP formulation is NP-complete. Furthermore, its assumptions are not realistic in practice. Reads can contain sequencing errors and hence they need not always appear preserved as substrings in the genome; and genomes typically are longer than the shortest superstring due to presence of repeats and nearly identical genic regions (called paralogs).

Among the more realistic models for genome assembly, graph theoretic formulations have been in the forefront. In broad terms, there are three distinct ways to model the problem (refer to Fig. 1):

(1) *Overlap graph*: Construct graph $G(V,E)$ from R, where each read is represented by a unique vertex in V and an edge is drawn between (r_i, r_j) iff there is a significant suffix–prefix overlap between r_i and r_j. *Overlap* is typically defined using a pairwise sequence alignment model (e.g., semi-global alignment) and related metrics to assess the quality of an alignment. Given G, the genome assembly problem can be reduced to the problem of finding a Hamiltonian Path, if one exists, provided that there were no breaks (called sequencing gaps) along the genome during sequencing. If there were gaps during sequencing, then one Hamiltonian Path is sought for every connected component in G (if it exists) and the genome can be recovered as an unordered set of pieces corresponding to the sequenced sections. This formulation using the overlap graph model has been shown to be NP-Hard.

(2) *De Bruijn graph:* Let a *k-mer* denote a string of length k. Construct a De Bruijn graph, where the vertex set is the set of all k-mers contained inside the reads of R; and a directed edge is drawn from vertices i to j, iff the $k-1$ length suffix of the k-mer i is identical to the $k-1$ length prefix of the k-mer j. Put another way, each edge in E represents a unique $k+1$-mer that is found in at least one of the reads in R. Given a De Bruijn graph G, genome assembly is the problem of finding a shortest Euler tour in which each read is represented by a sub-path in the tour. While finding an Euler tour is polynomially solvable, the optimization problem is still NP-Hard by reduction from SSP.

(3) *String graph*: This model is a variant of the overlap graph in which the edges represent reads, and vertices represent branching of end-to-end overlaps of the adjoining reads. For example, if a read r_i overlaps with both r_j and r_k then it is represented by a vertex



Genome Assembly. Fig. 1 Illustration of the three different models for genome assembly. The *top* section of the figure shows a hypothetical unknown genome and the reads sequenced from it. The *dotted box* shows a repetitive region within the genome. In the overlap graph, *solid lines* indicate true overlaps and *dotted lines* show overlaps induced by the presence of the repeat. The De Bruijn graph shows the graph for an arbitrary section involving three reads. The string graph shows an example of a branching vertex with multiple possible entry and exit paths

which has an inbound edge corresponding to r_i and two outbound edges representing r_j and r_k. Furthermore, the graph is pruned off transitively inferable edges – e.g., if both r_i and r_j overlaps with r_k, whereas r_i also overlaps with r_j, then the overlap from r_j to r_k is inferable and is therefore removed. Given a string graph G, the assembly problem becomes one that is equivalent of finding a constrained least cost Chinese Postman tour of G. This formulation has also been shown to be NP-Hard.

All the three formulations incorporate features to handle sequencing errors. In the overlap graph model, the pairwise alignment formulation used to define edges automatically takes into account character substitutions, insertions, and deletions. In the other two models, an "error correction" procedure is run as a preprocessing step to mask off anomalous portions of the graph prior to performing the assembly tours.

Parallelization for the Overlap Graph Model

Algorithms that use the overlap graph model deploy a three-stage assembly approach of *overlap–layout–consensus*. In the first stage, the overlap graph is constructed by performing all-against-all pairwise comparison of the reads in R. As a result, a layout for the assembly is prepared using the pairwise overlaps and in the third stage, a multiple sequence alignment (MSA) of the reads is performed as dictated by the layout. Given the large number of reads generated in a typical sequencing project, the overlap computation phase dominates the run-time and hence is the primary target for parallelization and performance optimization.

A brute-force way of implementing this step will be to compare all $\binom{m}{2}$ possible pairs. This approach can also be parallelized easily, as individual alignment tasks can be distributed in a load-balanced fashion given the uniformity expected in the read lengths. However, an implementation may not be practically feasible for larger number of reads due to the quadratic increase in alignment workload. Each alignment task executed serially could take milliseconds of run-time. In fact, for the assembly problem, the nature of sampling done during sequencing guarantees that performing all-against-all comparisons is destined to be highly wasteful. This is because of the random shotgun approach to sequencing which is expected to sample roughly equal number of reads covering each genomic base and it is typically only reads that originate from the same locus that tend to overlap with one another, with the exception of reads from repetitive regions of the genome.

A more scalable approach to detect pairwise overlaps is to first shortlist pairs of sequences based on the presence of sufficiently long exact matches (i.e., a filter built on a necessary-but-not-a-sufficient condition) and then compute pairwise comparisons only on them. Methods vary in the manner in which these "promising pairs" are shortlisted. One way of generating promising pairs is to use a string lookup table data structure that is built to record all k-mer occurrences within the reads in linear time. Sequence pairs that share one or more k-mers can be subsequently considered for alignment-based evaluation. While this approach supports a simple implementation, it poses a few notable limitations. The size complexity of the lookup table data structure contains an exponential term $O(|\Sigma|^k)$. This restricts the value of k in practice; for DNA alphabet, physical memory constraints demand that $k \leq 15$. Whereas, reads could share arbitrarily long matches, especially if the sequencing error rates are low. Another disadvantage of the lookup table is that it is only suited to detect pairs with fixed length matches. An exact match of an arbitrary length q will reveal itself as $q - k + 1$ smaller k-mer matches, thus increasing computation cost. Furthermore, the distribution of entries within the lookup table is highly data dependent and could scatter the same sequence pair in different locations of the lookup table, making it difficult for parallelization on distributed memory machines. Methods that use lookup tables simply use a high end shared memory machine for serial generation and storage, and focus on parallelizing only the pairwise sequence comparison step.

An alternative albeit more efficient way of enumerating promising pairs based on exact matches is to use a string indexing data structure such as suffix tree or suffix array that will allow detection of *variable* length matches. A match α between two reads r_i and r_j is called *left-maximal* (alternatively, *right-maximal*) if the characters preceding (alternatively, following) it in both strings, if any, are different. A match α between reads r_i and r_j is said to be a *maximal match* between the two reads if it is both left- and right-maximal. Pairs containing maximal matches of a minimum length cutoff, say

ψ, can be detected in optimal run-time and linear space using a suffix tree data structure. A generalized suffix tree of a set of strings is the compacted trie of all suffixes in the strings. The pair enumeration algorithm first builds a generalized suffix tree over all reads in R and then navigates the tree in a bottom-up order.

Parallel Generalized Suffix Tree Construction

For constructing a suffix tree, there are several linear time construction serial algorithms and optimal algorithms for the CRCW-PRAM model. However, optimal construction under the distributed memory machine model, where memory per processor is assumed to be too small to store the input sequences in R, is an open problem. Of the several approaches that have been studied, the following approach is notable as it has demonstrated linear scaling to thousands of processors on distributed memory supercomputers.

The major steps of the underlying algorithm are as follows. Each step is a parallel step. The steps marked as *comm* are communication-bound and the steps marked *comp* are computation-bound.

S1. *(comp)* Load R from I/O in a distributed fashion such that each processor receives $O\left(\frac{n}{p}\right)$ characters and no read is split across processor boundaries.

S2. *(comp)* Slide a window of length $k \leq \psi$ over the set of local reads and bucket sort locally all suffixes of reads based on their k length prefix. Note that for DNA alphabet, even a value as small as 10 for k is expected to create over a million buckets, sufficient to support parallel distribution. An alternative to local generation of buckets is to have a master–worker paradigm where the reads are scanned in small-sized batches and have a dedicated master distribute batches to workers in a load-balanced fashion.

S3. *(comm)* Parallel sort the buckets such that each processor receives approximately $\sim\frac{n}{p}$ suffixes and no bucket is split across processor boundaries. While the latter cannot be theoretically guaranteed, real-world genomic data sets typically distribute the suffixes uniformly across buckets thereby virtually ensuring that no bucket exceeds the $O\left(\frac{n}{p}\right)$ bound. If a bucket size becomes too big to fit in the local memory, then an alternative strategy can be deployed whereby the prefix length for bucketing is iteratively extended until the size of the bucket fits in the local memory.

S4. *(comm, comp)* Each bucket in the output corresponds to a unique subtree in the generalized suffix tree rooted exactly k characters below the root. The subtree corresponding to each bucket is then locally constructed using a recursive bucket sort–based method that compares characters between suffixes. However, this step cannot be done strictly locally because it needs the sequences of reads whose suffixes are being compared. This can be achieved by building the local set of subtrees in small-sized batches and performing one round of communication using *Alltoallv()* to load the reads required to build each batch from other processors. In order to reduce the overhead due to read fetches, observe that the aggregate memory on a relatively small group of processors is typically sufficient to accommodate the read set for most inputs in practice. For example, even a set of 100 million reads each of length 1Kbp needs only of the order of 100 GB. Even assuming 1 GB per processor, this means the read set will fit within 100 processors. Higher number of processors is required only to speedup computation. One could take advantage of this observation by partitioning the processor space into subgroups of fixed, smaller size, determined by the input size, and then have the on-the-fly read fetches to seek data from processors from within each subgroup. This improved scheme could distribute the load of any potential hot spots and also the overall communication could be faster as the collective transportation primitive now operates on a smaller number of processors.

The above approach can be implemented to run in $O\left(\frac{n\ell}{p}\right)$ time and $O\left(\frac{n}{p}\right)$ space, where ℓ is the mean read length. The output of the algorithm is a distributed representation of the generalized suffix tree, with each processor storing roughly $\frac{n}{p}$ suffixes (or leaves in the tree).

The *cluster-then-assemble* Approach

A potential stumbling block in the overlap–layout–consensus approach is the large amount of memory required to store and retrieve the pairwise overlaps so that they can be used for generating an assembly layout. For genomes which have been sequenced at a uniform

coverage c, the expectation is that each base along the genome is covered by c reads on an average, thereby implying $\binom{c}{2}$ overlapping read pairs for every genomic base. However, this theoretical linear expectation holds for only a fraction of the target genome, whereas factors such as genomic repeats and oversampled genic regions could increase the number of overlapping pairs arbitrarily beyond the expected level. An example case in point is the gene-enriched sequencing of the highly repetitive ~2.5 billion bp maize genome.

One way to overcome this scalability bottleneck is to use clustering as a preprocessing step to assembly. This approach, referred to as *cluster-then-assemble*, builds upon the assumption that any genome-scale sequencing effort is likely to undersample the genome, leaving out sequencing gaps along the genome length and thereby allowing the assemblers to reconstruct the genome in pieces – one piece for every contiguous stretch of the genome (aka "contig" or "genomic island") that is fully sampled through sequencing. This assumption is not unrealistic as tens of thousands of sequencing gaps have always resulted in almost all eukaryotic genomes sequenced so far using the WGS approach. The cluster-then-assemble takes advantage of this assumption as follows: The set of m reads is first partitioned into groups using a single-linkage transitive closure clustering method, such that there exists a sequence of overlapping pairs connecting every read to every other read in the same cluster. Ideally, each cluster should correspond to one genomic island, although the presence of genomic repeats could collapse reads belonging to different islands together. Once clustered, each cluster represents an independent subproblem for assembly, thereby breaking a large problem with m reads into numerous, disjoint subproblems of significantly reduced size small enough to fit in a serial computer. In practice, tens of thousands of clusters are generated making the approach highly suited for trivial parallelization after clustering.

The primary challenge is to implement the clustering step in parallel, in a time- and space-efficient manner. In graph-theoretic terms, the output produced by a transitive closure clustering is representative of connected components in the overlap graph, thereby allowing the problem to be reduced to one of connected component detection. However, generating the entire graph G prior to detection would contradict the purpose of clustering from a space complexity standpoint.

Figure 2 outlines an algorithm to perform sequence clustering. The algorithm can be described as follows: Initialize each read in a cluster of its own. In an iterative process, generate promising pairs based on maximal matches in a non-increasing order of their lengths using suffix trees (as explained in the previous section). Before assigning a promising pair for further alignment processing, a check is made to see whether the constituent reads are in the same cluster. If they are part of the same cluster already, then the pair is discarded; otherwise it

Algorithm 1 *Read Clustering*

Input: Read set $R = \{r_1, r_2, \ldots r_m\}$
Output: A partition $C = \{C_1, C_2, \ldots C_k\}$ **of** R**,** $1 \leq k \leq m$

1. Initialize Clusters: $\Rightarrow (master)$
   $C \leftarrow \{\ \{r_i\} \mid 1 \leq i \leq m\}$
2. FOR each pair (r_i, r_j) with a maximal match of length $\geq \psi$ generated in non-increasing order of maximal match length $\Rightarrow (worker)$
   $C_p \leftarrow Find(r_i)$ $\Rightarrow (master)$
   $C_q \leftarrow Find(r_j)$ $\Rightarrow (master)$
   IF $C_p \neq C_q$ THEN $\Rightarrow (master)$
   overlap quality $\leftarrow Align(r_i, r_j)$ $\Rightarrow (worker)$
   IF overlap quality is significant THEN $\Rightarrow (master)$
   $Union(C_p, C_q)$ $\Rightarrow (worker)$
3. Output C $\Rightarrow (master)$

Genome Assembly. Fig. 2 Pseudocode for a read clustering algorithm suited for parallelization based on the master-worker model. The site of computation is shown in brackets. Operations on the set of clusters are performed using the Union-Find data structure

is aligned. If the alignment results show a satisfactory overlap (based on user-defined alignment parameters), then merge the clusters containing both reads into one larger cluster. The process is repeated until all promising pairs are exhausted or until no more merges are possible. Using the union–find data structure would ensure the *Find* and *Union* calls to run in amortized time proportional to the Inverse Ackerman function – a small constant for all practical inputs.

There are several advantages to this clustering strategy. Clustering can be achieved in at most $m-1$ merging steps. Checking if a read pair is already clustered before alignment is aimed at reducing the number of pairs aligned. Generating promising pairs in a non-increasing order of their maximal match lengths is a heuristic that could help identify pairs that are more likely to succeed the alignment test sooner during execution. Furthermore, promising pairs are processed as they are generated, obviating the need to store them. This coupled with the use of the suffix tree data structure implies an $O(n)$ serial space complexity for the clustering algorithm.

The parallel algorithm can be implemented using a master–worker paradigm. A dedicated master processor can be responsible for initializing and maintaining the clusters and also for distributing alignment workload to the workers in a load-balanced fashion. The workers at first can generate a distributed representation of the generalized suffix tree in parallel (as explained in the previous section). Subsequently, they can generate promising pairs from their local portion of the tree and send them to the master. To reduce communication overheads, pairs can be sent in arbitrary-sized batches as demanded by the situation of the work queue buffer at the master. The master processor can check the pairs against the current set of clusters, filter out pairs that do not need alignment, and add only those pairs that need alignment to its work queue buffer. The pairs in the work queue buffer can be redistributed to workers in fixed size batches, to have the workers compute alignments and send back the results. Communication overheads can be masked by overlapping alignment computation with communication waits using non-blocking calls. The PaCE software suite implements the above parallel algorithm, and it has demonstrated linear scaling to thousands of processors on a distributed memory supercomputer.

Short Read Assembly

The landscape of genome assembly tools has significantly transformed itself over the last few years with the advent of next-generation sequencing technologies. A plethora of new-generation assemblers that collectively operate under the banner of "short read assemblers" is continuing to emerge. Broadly speaking, these tools can be grouped into two categories: (1) those that follow or extend from the classical overlap graph model; and (2) those that deploy either the De Bruijn graph or the string graph formulation. The former category includes programs such as Edena, Newbler, PE-Assembler, QSRA, SHARCGS, SSAKE, and VCAKE. The programs that fall under the latter category are largely inspired by two approaches – the De Bruijn graph formulation used in the EULER assembler; and the string graph formulation proposed by Myers. As of this writing, these programs include EULER-SR, ALLPATHS, Velvet, SOAPdenovo, ABySS, and YAGA. Either of these lists is likely to expand in the coming years, as new implementations of short read assemblers are continuing to emerge as are new technologies.

In principle, the techniques developed for assembling Sanger reads do not suit direct application for short read assembly due to a combination of factors that include shorter read length, higher sequencing coverage, an increased reliance on pair-end libraries, and idiosyncrasies in sequencing errors.

A shorter read length implies that the method has to be more sensitive to differences to avoid potential mis-assemblies. This also means providing a robust support for incorporating the knowledge available from pair-end libraries. In case of next-generation sequencing, pair-end libraries are typically available for different clone insert sizes, which translates to a need to implement multiple sets of distance constraints.

A high sequencing coverage introduces complexity at two levels. The number of short reads to assemble increases linearly with increased coverage, and this number can easily reach hundreds of millions even for modest-sized genomes because of shorter read length. For instance, a mid-size genome such as that of Arabidopsis (~115 Mbp) or fruit fly (~120 Mbp) sequenced at 100x coverage either using Illumina or SOLiD would generate a couple of hundred million reads. Secondly, the average number of overlapping read

pairs at every given genomic location is expected to grow quadratically $\left(\propto \binom{c}{2}\right)$ with the coverage depth (c). This particularly affects the time and memory scalability of methods that use the overlap graph model not only because they operate at the level of pairwise read overlaps, but also because many tools assume that such overlaps can be stored and retrieved from local memory. Consequently, such methods take between 6–10 h and several gigabytes of memory even for assembling small bacterial genomes. From a parallelism perspective, a high coverage in sampling could also potentially mean fewer sequencing gaps, as the probability of a genomic base being captured by a read improves with coverage by the Lander–Waterman model. Therefore, divide-and-conquer–based techniques such as the cluster-then-assemble may not be as effective in breaking the initial problem size down.

Short read assemblers also have to deal with technology specific errors. The error rates associated with next-generation technologies are typically in the 1–5% range, and cannot be ignored as differentiating them from real differences could be key to capturing natural variations such as near identical paralogs.

In the current suite of tools specifically built for short read assembly, only a handful of tools support some degree of parallelism. These tools include PE-Assembler, ABySS, and YAGA. Others are serial tools that work on desktop computers and rely on high-memory nodes for larger inputs. Even the tools that support parallelism do so to varying degrees. PE-Assembler, which implements a variant of the overlap graph model, limits parallelism to node level and assumes that the number of parallel threads is small (< 10). It deploys a "seed-extend" strategy in which a subset of "reliable" reads are selected as seeds and other reads that overlap with each seed are incrementally added to extend and build into a consensus sequence in the $3'$ direction. Parallelism is supported by selecting multiple seeds and launching multiple threads that assume responsibility of extending these different reads in parallel. While the algorithm has the advantage of not having to build and manage large graphs, it performs a conservative extension primarily relying on pair-end data to collapse reads into contigs. From a parallelism perspective, the algorithm relies on shared memory access for managing the read set, which works well if the number of threads is very small. Furthermore, not all steps in the method are parallel and some steps are disk-bound. Experimental results show that the parallel efficiency drops drastically beyond three threads.

ABySS and YAGA are two parallel methods that implement the De Bruijn graph model and work for distributed memory computers. The De Bruijn and string graph formulations are conceptually better suited for short read assembly. These formulations allow the algorithm to operate at a read or read's subunit (i.e., k-mer) level rather than the pairwise overlap level. Repetitive regions manifest themselves in the form of special graph patterns, and error correction mechanism could detect and reconcile anomalous graph substructures prior to performing assembly tours, thereby reducing the chance of misassemblies. The assembly itself manifests in the form of graph traversals. Constructing these graphs and traversing them to produce assembly tours (although not guaranteed to be optimal due to intractability of the problem) are problems with efficient heuristic solutions on a serial computer. The methods ABySS and YAGA provide two different approaches to implement the different stages in parallel using De Bruijn graphs. While these methods differ in their underlying algorithmic details and in the degree offered for parallelism, the structural layout of their algorithms is similar consisting of these four major steps: (1) parallel graph construction; (2) error correction; (3) incorporation of distance constraints due to pair-end reads; and (4) assembly tour and output.

In what follows, approaches to parallelize each of these major steps are outlined, with the bulk of exposition closely mirroring the algorithm in YAGA because its algorithm more rigorously addresses parallelization for all the steps, and takes advantage of techniques that are more standard and well understood among distributed memory processing codes (e.g., sorting, list ranking). By contrast, parallelization is described only for the initial phase of graph construction in ABySS. Therefore, where appropriate, variations in the ABySS algorithmic approach will be highlighted in the text. Other than differences in their parallelization strategies, the two methods also differ particularly in the type of De Bruijn graph they construct (directed vs. bidirected) and their ability to handle multiple read lengths. Such details are omitted in an attempt to keep the text focused on the parallel aspects of the problem. An interested reader can refer to the individual papers for details

pertaining to the nuances of the assembly procedure and the output quality of these assemblies.

Parallel De Bruijn Graph Construction and Compaction

Given m reads in set R, the goal is to construct in parallel a distributed representation of the corresponding De Bruijn graph built out of k-mers, for a user-specified value of k. Note that, once a De Bruijn graph representation is generated, it can be transformed into a corresponding string graph representation by compressing paths whose label when concatenated spells out the characters in a read, and by accordingly introducing branch nodes that capture overlap continuation between adjoining reads. This can be done in a successive stage of graph compaction, a minor variant of which is described in the later part of this section. The computing model assumed is p processors (or equivalently, processes), each with access to a local RAM, and connected through a network interconnect and with access to a shared file system where the input is made available.

To construct the De Bruijn graph, the reads in R are initially partitioned and loaded in a distributed manner such that each processor receives $O\left(\frac{n}{p}\right)$ input characters. Let R_i refer to the subset of reads in processor p_i. Through a linear scan of the reads in R_i, each p_i enumerates the set of k-mers present in R_i. However, instead of storing each such k-mer as a vertex of the De Bruijn graph, the processor equivalently generates and stores the corresponding *edges* connecting those vertices in the graph. In other words, there is a bijection between the set of edges and the set of distinct $k+1$-mers in R_i. In this edge-centric representation, the vertex information connecting each edge is implicit and the count of reads containing a given $k+1$-mer is stored internally at that edge. Note that after this generation process, the same edge could be potentially generated at multiple processor locations. To detect and merge such duplicates, the algorithm simply performs a *parallel sort* of the edges using the $k+1$-mers as the key. Therefore, in one sorting step, a distributed representation of the De Bruijn graph is constructed. Standard parallel sort routines such as sample sort can be used here to ensure even redistribution of the edges across processors due to sorting.

An alternative to this edge-centric representation is a vertex-centric representation, which is followed in ABySS and in an earlier version of YAGA. Here, the k-mer set corresponding to each R_i is generated by the corresponding p_i. The vertices adjacent to a given vertex could be generated remotely by one or more processors. Therefore, this approach necessitates processors to communicate with one another in order to check the validity of all edges that could be theoretically drawn from its local set of vertices. While the number of such edge validation queries is bounded by 8 per vertex ($\{a, c, g, t\}$ on either strand orientation), the method runs the risk of generating false edges, e.g., if a $k-1$-mer, say α, occurs in exactly two places along the genome as $a\alpha c$ and $g\alpha t$, then the vertex-centric approach will erroneously generate edges for $a\alpha t$ and $g\alpha c$, which are $k+1$-mers nonexistent in the input. Besides this risk, care must be taken to guarantee an even redistribution of vertices among processors by the end of the construction process. For instance, a static allocation scheme in which a hash function is used to map each k-mer to a destination processor, as it is done in ABySS, runs the risk of producing unbalanced distribution as the k-mer concentration within reads is input dependent.

Graph compaction: Once a distributed edge-centric representation of the De Bruijn graph is generated, the next step is to simplify it by identifying maximal segments of simple paths (or "chains") and compact each of them into a single longer edge. Edges that belong to a given chain could be distributed and therefore this problem of removing chains becomes a two-step process: First, to detect the edges (or equivalently, the vertices) which are part of chains and then perform compaction on the individual chains. To mark the vertices that are part of some chain, assume without loss of generality that each edge is stored twice as $< u, v >$ and $< v, u >$. *Sorting* the edges in parallel by their first vertex it would bring all edges incident on a vertex together on some processor. Only those vertices that have a degree of two can be part of chains and such vertices can be easily marked with a special label. In the next step, the problem of compacting the vertices along each chain can be treated as a variant of *segmented parallel list ranking*. A distributed version of the list ranking algorithm can be used to calculate the relative distances of each marked vertex from the start and end of its respective chain. The same procedure can also be used to determine the

vertex identifiers of those two boundary vertices for each marked vertex. Compaction then follows through the operations of concatenating the edge labels along the chain at one of the terminal edges, removing all internal edges, and aggregating an average $k + 1$-mer frequency of the constituent edges along the chain. All of these operators are binary associative to allow being implemented using calls to a segmented parallel prefix routine. The output of this step is a distributed representation of the compacted graph.

Error Correction and Variation Detection

In the De Bruijn graph representation, errors due to sequencing manifest themselves as different subgraph motifs which could be detected and pruned. For ease of exposition, let us informally call an edge in the compacted De Bruijn graph as being "strongly supported" (alternatively, "weakly supported") if its average $k + 1$-mer frequency is relatively high (alternatively, low). Examples of motifs are follows: (1) *Tips* are weakly supported dead ends in the graph created due to a base miscall occurring in a read at one of its end positions. Because of compaction such tips occur as single edges branching out of a strongly supported path, and can be easily removed; (2) *Bubbles* are detours that provide alternative paths between two terminal vertices. Due to compaction, these detours will also be single edges. There are two types of bubbles – weakly supported bubbles are manifestations of single base miscall occurring internal to a read and need to be removed; whereas, bubbles that originate from the same vertex and are supported roughly to the same degree could be the result of natural variations such as near identical paralogs (i.e., copies of the same gene occurring at different genomic loci). Such bubbles need to be retained. (3) *Spurious links* connect two otherwise disparate paths in the graph. Weakly supported links are manifestations of erroneous $k + 1$-mers that happen to match the $k + 1$-mer present at a valid genomic locus, and they can be severed by examining the supports of the other two paths.

The first pass of error correction on the compacted graph could reveal new instances of motifs that can be pruned through iterative passes subsequently until no new instances are observed.

Incorporation of Distance Constraints Using Pair-End Information

The goal of this step is to map the information provided by read pairs that are linked by the pair-end library onto the compacted and error corrected De Bruijn graph. Once mapped, this information can be used to guide the assembly tour of the graph consistent (to the extent possible) with the distance constraints imposed by the pair-end information. To appreciate the value added by this step to the assembly procedure, recall that the pair-end information consists of a list of read pairs of the form $< r_i, r_j >$ that have originated from the same clonal insert during sequencing. In genomic distance parlance, this implies that the number of bases separating the two reads is bounded by a minimum and maximum. Also recall that an assembly from a De Bruijn graph corresponds to a tour of the graph (possibly multiple tours if there were gaps in the original sequencing). Now consider a scenario where a path in the De Bruijn graph branches into multiple separate subpaths. A correct assembly tour would have to decide which of those branches reflect the sequence of characters along the unknown genome. It should be easy to see how pair-end information can be used to resolve such situations.

To incorporate the information provided in the form of read pairs by such pair-end libraries, the YAGA algorithm uses a *cluster summarization* procedure, which can be outlined as follows: First, the list of read pairs provided as input by the pair-end information is delineated into a corresponding list of constituent $k + 1$-mer pairs. Note that two $k + 1$-mers from two reads of a pair can map to two different edges on the distributed graph. Furthermore, different positions along the same edge could be paired with positions emanating from different edges. To this effect, the algorithm attempts to compute a grouping of edge pairs based on their best alignment with the imposed distance constraints. To achieve this, an observed distance interval is computed between every edge pair on the graph linked by a read pair, and then overlapping intervals that capture roughly the same distances along the same orientation are incrementally clustered using a greedy heuristic. This last step is achieved using a two-phase clustering step that primarily relies on several rounds of *parallel sorting* tuples containing edge pair and interval distance information. The formal details pertaining to this algorithmic step has been omitted here for brevity.

Completing the Assembly Tour

An important offshoot of the above summarization procedure is that redundant distance information captured by edge pairs in the same cluster can be removed, thereby allowing significant compression in the level of information needed to store relative to the original graph. This compression, in most practical cases, would allow for the overall tour to be performed sequentially on a single node. The serial assembly touring procedure begins by using edges that have significantly longer edge labels than the original read length as "seeds," and then by extending them in both directions as guided by the pair-end summarization traversal constraints.

The YAGA assembler has demonstrated scaling on 512 IBM BlueGene/L processors and performs assembly of over a billion synthetically generated reads in under 2 h. The run-time is dominated by the pair-end cluster summarization phase. As of this writing, performance-related information is not available for ABySS.

Future Trends

Genome sequencing and assembly is an actively pursued, constantly evolving branch of bioinformatics. Technological advancements in high-throughput sequencing have fueled algorithmic innovation and along with a compelling need to harness new computing paradigms. With the adoption of ever faster, cheaper, and massively parallel sequencing machines, this application domain is becoming increasingly data- and compute-intensive. Consequently, parallel processing is destined to play a critical role in genomic discovery.

Numerous large-scale sequencing projects including the 2001 assembly of the human genome to the most recent 2009 assembly of the maize genome have benefited from the use of parallel processing, although in different ad hoc ways. Heterogeneous clusters comprising of a mixture of a few high-end shared memory machines along with numerous compute nodes have been used to "farm" out tasks and accelerate the overlap computation phase in particular. This is justified because nearly all of these large-scale projects used the more traditional Sanger sequencing. A few special-purpose projects such as the 2005 maize gene-enriched sequencing used more strongly coupled parallel codes such as PaCE. However, with an aggressive adoption of next-generation sequencing for genome sequencing and more complex genomes in the pipeline for sequencing (e.g., wheat, pine, metagenomic communities), this scenario is about to change, and more strongly coupled parallel codes are expected to become part of the mainstream computing in genome assembly.

In addition to *de novo* sequencing, next-generation sequencing is also increasingly being used in *genome resequencing* projects, where the goal is to assemble a genome of a particular variant strain (or subspecies) of an already sequenced genome. The type of computation that originates is significantly different from that of *de novo* assembly. For resequencing, the reads generated from a new strain are compared against a fully assembled sequence of a reference strain. This process, sometimes called *read mapping*, only requires comparison of reads against a much larger reference. Approaches that capitalize on advanced string data structures such as suffix trees are likely to play an active role in these algorithms. Also, as this branch of science becomes more data-intensive, reaching the petascale range, different parallel paradigms such as MapReduce need to be explored, in addition to distributed memory and shared memory models. The developments in genome sequencing can be carried over to other related applications that also involve large-scale sequence analysis, e.g., in transcriptomics, metagenomics, and proteomics.

Fine-grain parallelism in the area of string matching and sequence alignment has been an active pursued topic over the last decade and there are numerous hardware accelerators for performing sequence alignment on various platforms including General Purpose Graphical Processing Units, Cell Broadband Engine, Field-Programmable Gate Arrays, and Multicores. These advances are yet to take their place in mainstream sequencing projects.

Genome sequencing is at the cusp of revolutionary possibilities. With a rapidly advancing technology base, the possibility of realizing landmark goals such as personalized medicine and "$1,000 genome" do not look distant or far-fetched. The well-advertised $10 million Archon Genomics X PRIZE will be awarded to the first team that sequences 100 human genomes in 10 days, at a recurring cost of no more than $10,000 per genome. In 2010, the cost for sequencing a human genome plummeted below $10,000 using technologies from Illumina and SOLiD. Going past these next-generation technologies, however, companies such as

Pacific Biosciences are now releasing a third-generation ("gen-3") sequencer that uses an impressive approach called single-molecule sequencing (or "SMS") [9], and have proclaimed grand goals such as sequencing a human genome in 15 min for less than $100 by 2014. These are exciting times for genomics, and the field is likely to continue serving as a rich reservoir for new problems that pose interesting compute- and data-intensive challenges which can be addressed only through a comprehensive embrace of parallel computing.

Related Entries

►Homology to Sequence Alignment, From
►Suffix Trees

Bibliographic Notes and Further Reading

Even though DNA sequencing technologies have been available since the late 1970s, it was not until the 1990s that they were applied at a genome scale. The first genome to be fully sequenced and assembled was the ~1.8 Mbp bacterial genome of *H. influenzae* in 1995 [8]. The sequencing of the more complex, ~3 billion bp long human genome followed [27]. Several other notable large-scale sequencing initiatives followed in the new millennium including that of chimpanzee, rice, and maize (to cite a few examples). All of these used either the WGS strategy or hierarchical strategy coupled with Sanger sequencing, and their assemblies were performed using programs that followed the overlap–layout–consensus model. The National Center for Biotechnology Information (NCBI) (http://www.ncbi.nlm.nih.gov) maintains a comprehensive database of all sequenced genomes. More than a dozen programs exist to perform genome assembly under the overlap–layout–consensus model. Notable examples include PCAP [12], Phrap (http://www.phrap.org/), Arachne [2], Celera [22], and TIGR assembler [26]. For a detailed review of fragment assembly algorithms, refer to [7, 24]. The parallel algorithm that uses the cluster-then-assemble approach along with the suffix tree data structure for assembly was implemented in a program called PaCE and was first described in [16] in the context of clustering Expressed Sequence Tag data and then later adapted for genome assembly [17]. Experiments conducted as part of the maize genome sequencing consortium demonstrated scaling of this method to over a million reads generated from gene-enriched fractions of the maize genome on a 1,024 node BlueGene/L supercomputer [17]. The parallel suffix tree construction algorithm described in this entry was first described in [16] and a variant of this method was later presented in [11]. An optimal algorithm to detect maximal matching pairs of reads in parallel using the suffix tree data structure is presented in [16].

The NP-completeness of the Shortest Superstring Problem (SSP) was shown by Gallant et al. [10]. The De Bruijn graph formulation for genome assembly was first introduced by Idury and Waterman [13] in the context of a sequencing technique called sequencing-by-hybridization, and later extended to WGS based approaches in the EULER program by Pevzner et al. [23]. The string graph formulation was developed by Myers [21]. The proof of NP-Hardness for the overlap–layout–consensus is due to Kececioglu and Myers [18]. The proofs of NP-Hardness for the De Bruijn and string graphs models of genome assembly are due to Medvedev et al. [20].

Since the later part of 2000s, various next-generation sequencing technologies such as Roche 454 (http://www.genome-sequencing.com/), SOLiD (http://www.appliedbiosystems.com/), Illumina (http://www.illumina.com/), and HeliScope (http://www.helicosbio.com/) have emerged along side serial assemblers. A "third" generation of machines that promise a brand new way of sequencing (by single-molecule sequencing) are also on their way (e.g., Pacific Biosciences (http://www.pacificbiosciences.com/)). Consequently, the development of short read assemblers continue to be in hot pursuit. Edena, Newbler (http://www.454.com), PE-Assembler [1], QSRA [3], SHARCGS [6], SSAKE [28], and VCAKE [15] are all examples of programs that operate using the overlap graph model. EULER-SR [5], ALLPATHS [4], Velvet [29], SOAPdenovo [19], ABySS [25], and YAGA [14] are programs that use the De Bruijn graph formulation. Of these tools, PE-Assembler, ABySS, and YAGA are parallel implementations, although to varying degrees as described in the main text.

Acknowledgment

Study supported by NSF grant IIS-0916463.

Bibliography

1. Ariyaratne P, Sung W (2011) PE-assembler: de novo assembler using short paired-end reads. Bioinformatics 27(2):167–174
2. Batzoglou S, Jaffe DB, Stanley K, Butler J et al (2002) ARACHNE: a whole-genome shotgun assembler. Genome Res 12(1):177–189
3. Bryant D, Wong W, Mockler T (2009) QSRA – a quality-value guided de novo short read assembler. BMC Bioinform 10(1):69
4. Butler J, MacCallum L, Kleber M, Shlyakhter IA et al (2008) ALLPATHS: de novo assembly of whole-genome shotgun microreads. Genome Res 18:810–820
5. Chaisson MJ, Pevzner PA (2008) Short read fragment assembly of bacterial genomes. Genome Res 18:324–330
6. Dohm J, Lottaz C, Borodina T, Himmelbaurer H (2007) SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. Genome Res 17(11):1679–1706
7. Emrich S, Kalyanaraman A, Aluru S (2005) Chapter 13: algorithms for large-scale clustering and assembly of biological sequence data. In: Handbook of computational molecular biology. CRC Press, Boca Raton
8. Fleischmann R, Adams M, White O, Clayton R et al (1995) Whole-genome random sequencing and assembly of Haemophilus influenzae rd. Science 269(5223):496–512
9. Flusberg BA, Webster DR, Lee JH, Travers KJ et al (2010) Direct detection of DNA methylation during single-molecule, real-time sequencing. Nat Methods 7:461–465
10. Gallant J, Maier D, Storer J (1980) On finding minimal length superstrings. J Comput Syst Sci 20:50–58
11. Ghoting A, Makarychev K (2009) Indexing genomic sequences on the IBM blue gene. In: Proceedings ACM/IEEE conference on supercomputing. Portland
12. Huang X, Wang J, Aluru S, Yang S, Hiller L (2003) PCAP: a whole-genome assembly program. Genome Res 13:2164–2170
13. Idury RM, Waterman MS (1995) A new algorithm for DNA sequence assembly. J Comput Biol 2(2):291–306
14. Jackson BG, Regennitter M, Yang X, Schnable PS, Aluru S (2010) Parallel de novo assembly of large genomes from high-throughput short reads. In: IEEE international symposium on parallel distributed processing, pp 1–10
15. Jeck W, Reinhardt J, Baltrus D, Hickenbotham M et al (2007) Extending assembly of short DNA sequences to handle error. Bioinformatics 23:2942–2944
16. Kalyanaraman A, Aluru S, Brendel V, Kothari S (2003) Space and time efficient parallel algorithms and software for EST clustering. IEEE Trans Parallel Distrib Syst 14(12):1209–1221
17. Kalyanaraman A, Emrich SJ, Schnable PS, Aluru S (2007) Assembling genomes on large-scale parallel computers. J Parallel Distrib Comput 67(12):1240–1255
18. Kececioglu J, Myers E (1995) Combinatorial algorithms for DNA sequence assembly. Algorithmica 13(1–2):7–51
19. Li R, Zhu H, Ruan J, Qian W et al (2010) De novo assembly of human genomes with massively parallel short read sequencing. Genome Res 20(2):265–272
20. Medvedev P, Georgiou K, Myers G, Brudno M (2007) Computability of models for sequence assembly. Lecture notes in computer science, vol 4645. Springer, Heidelberg, pp 289–301
21. Myers EW (2005) The fragment assembly string graph. Bioinformatics, 21(Suppl 2):ii79–ii85
22. Myers EW, Sutton GG, Delcher AL, Dew IM et al (2000) A Whole-Genome assembly of drosophila. Science 287(5461):2196–2204
23. Pevzner PA, Tang H, Waterman M (2001) An eulerian path approach to DNA fragment assembly. In: Proceedings of the national academy of sciences of the United States of America, vol 98, pp 9748–9753
24. Pop M (2009) Genome assembly reborn: recent computational challenges. Briefings in Bioinformatics 10(4):354–366
25. Simpson J, Wong K, Jackman S, Schein J et al (2009) ABySS: a parallel assembler for short read sequence data. Genome Res 19:1117–1123
26. Sutton GG, White O, Adams MD, Kerlavage AR (2011) TIGR assembler: a new tool for assembling large shotgun sequencing projects. Genome Sci Technol 1(1):9–19
27. Venter C, Adams MD, Myers EW, Li P et al (2001) The sequence of the human genome. Science 291(5507):1304–1351
28. Warren P, Sutton G, Holt R (2006) Assembling millions of short DNA sequences using SSAKE. Bioinformatics 23:500–501
29. Zerbino DR, Velvet BE (2008) Algorithms for de novo short read assembly using de bruijn graphs. Genome Res 18:821–829

Genome Sequencing

►Genome Assembly

3GIO

►PCI Express

Glasgow Parallel Haskell (GpH)

Kevin Hammond
University of St. Andrews, St. Andrews, UK

Synonyms

GpH (Glasgow Parallel Haskell)

Definition

Glasgow Parallel Haskell (GpH) is a simple parallel dialect of the purely functional programming language, Haskell. It uses a *semi-explicit* model of parallelism, where possible parallel threads are marked by the programmer, and a sophisticated runtime system then

decides on the timing of thread creation, allocation to processing elements, migration. There have been several implementations of GpH, covering platforms ranging from single multicore machines through to computational grids or clouds. The best known of these is the GUM implementation that targets both shared-memory and distributed-memory systems using a sophisticated virtual shared-memory abstraction built over a common message-passing layer, but there is a new implementation, GHC-SMP, which directly exploits shared-memory systems, and which targets multicore architectures.

Discussion

Purely Functional Languages and Parallelism

Because of the absence of side effects in purely functional languages, it is relatively straightforward to identify computations that can be run in parallel: any sub-expression can be evaluated by a dedicated parallel task. For example in the following very simple function definition, each of the two arguments to the addition operation can be evaluated in parallel.

```
f x = fibonacci x + factorial x
```

This property was already realized by the mid-1980s, when there was a surge of interest both in parallel evaluation in general, and in the novel architectural designs that it was believed could overcome the sequential "von-Neumann bottleneck." In fact, the main issue in a purely functional language is not extracting enough parallelism – it is not unusual for even a short-running program to produce many tens of thousands of parallel threads – but is rather one of identifying sufficiently large-grained parallel tasks. If this is not done, then thread creation and communication overheads quickly eliminate any benefit that can be obtained from parallel execution. This is especially important on conventional processor architectures, which historically provided little, if anything, in the way of hardware support for parallel execution. The response of some parallel functional language designers has therefore been to provide very explicit parallelism mechanisms. While this usually avoids the problem of excessive parallelism, it places a significant burden on the programmer, who must understand the details of parallel execution as well as the application domain, and it can lead to code that is specialized to a specific parallel architecture, or class of architectures. It also violates a key design principle for most functional languages, which is to provide as much isolation as possible from the underlying implementation. GpH is therefore designed to allow programmers to provide information about parallel execution while delegating issues of placement, communication, etc. to the runtime system.

History and Development of GpH

Glasgow Parallel Haskell (GpH) was first defined in 1990. GpH was designed to be a simple parallel extension to the then-new nonstrict, purely functional language Haskell [10], adding only two constructs to sequential Haskell: *par* and *seq*. Unlike most earlier *lazy* functional languages, Haskell was always intended to be parallelizable. The use of the term "non-strict" rather than lazy reflects this: while lazy evaluation is inherently sequential, since it fixes a specific evaluation order for sub-expressions, under Haskell's non-strict evaluation model, any number of sub-expressions can be evaluated in parallel provided that they are needed by the result of the program.

The original GpH implementation targeted the GRIP novel parallel architecture, using direct calls to low-level GRIP communications primitives. GRIP was a shared-memory machine, using custom microcoded "intelligent memory units" to share global program data between off-the-shelf processing elements (Motorola 68020 processors, each with a private 4 MB memory), developed in an Alvey research project which ran from 1985 to 1988. Initially, GpH built on the prototype **Haskell** compiler developed by Hammond and Peyton Jones in 1989. It was subsequently ported to the Glasgow Haskell Compiler, GHC [9], that was developed at Glasgow University from 1991 onward, and which is now the de-facto standard compiler for Haskell (since the focus of the maintenance effort was moved to Microsoft Research in 1998, GHC has also become known as the Glorious Haskell Compiler). In 1994, the communications library was redesigned to give what became the highly portable GUM implementation [15]. This allowed a single parallel implementation to target both commercial shared-memory systems and the then-emerging class of cost-effective loosely coupled networks of workstations. Initially, GUM targeted

system-specific communication libraries, but it was subsequently ported to PVM. There are now UDP, PVM, MPI, and MPICH-G2 instances of GUM, as well as system-specific implementations, and the same GUM implementation runs on multicore machines, shared-memory systems, workstation clusters, computational grids, and is being ported to large-scale high-performance systems, such as the 22,656-core HECToR system at the Edinburgh Parallel Computer Centre. Key parts of the GUM implementation have also been used to implement the **Eden** [5] parallel dialect of Haskell, and GpH is being incorporated into the latest mainstream version of GHC.

Although the two parallelism primitives that GpH uses are very simple, it became clear that they could be packaged using higher-order functions to give much higher-level parallel abstractions, such as parallel pipelines, data-parallelism, etc. This led ultimately to the development of *evaluation strategies* [14]: high-level parallel structures that are built from the basic *par* and *seq* primitives using standard higher-order functions. Because they are built from simple components and standard language technology, evaluation strategies are highly flexible: they can be easily composed or nested; the parallelism structure can change dynamically; and the applications programmer can define new strategies on an as-need basis, while still using standard strategies.

The GpH Model of Parallelism

GpH is unusual in using only two parallelism primitives: *par* and *seq*. The design of the *par* primitive dates back to the late 1980s [8], where it was used in a parallel implementation of the Lazy ML (LML) compiler. The primitive, higher-order, function, *par* is used by the programmer to mark a sub-expression as being suitable for parallel evaluation. For example, a variant of the function `f` above can be parallelized using two instances of *par*.

```
f x =
   let r1 = fibonacci x;
   r2 = factorial x in
   let result = (r1, r2) in
   r1 `par` (r2 `par` result)
```

`r1` and `r2` can now both be evaluated in parallel with the construction of the result pair `(r1, r2)`. There is no need to specify any explicit communication, since the results of each computation are shared through the variables `r1` and `r2`. The runtime system also decides on issues such as when a thread is created, where it is placed, how much data is communicated, etc. Very importantly, it can also decide *whether* a thread is created. The parallelism model is thus *semi-explicit*: the programmer marks possible sites of parallelism, but the runtime system takes responsibility for the underlying parallel control based on information about system load, etc. This approach therefore eliminates significant difficulties that are commonly experienced with more explicit parallel approaches, such as raw MPI. The programmer needs to make sure there is enough scope for parallelism, but does not need to worry about issues of deadlock, communication, throttling, load-balancing, etc. In the example above, it is likely that only one of `r1` or `r2` (or neither) will actually be evaluated in parallel, since the current thread will probably need both their values. Which of `r1` or `r2` is evaluated first by the original thread will, however, depend on the context in which it is called. It is therefore left unspecified to avoid unnecessary sequentialization.

Lazy Thread Creation

Several different versions of the *par* function have been described in the literature. The version used in GpH is asymmetric in that it marks its first argument as being suitable for possible execution (the expression is *sparked*), while continuing sequential execution of its second argument, which forms the result of the expression. For example, in `par s e`, the expression `s` will be sparked for possible parallel evaluation, and the value of `e` will be returned as the result of the current thread. Since the result expression is always evaluated by the sparking thread, and since there can be no side effects, it follows that it is completely safe to ignore any spark. That is, unlike many parallel systems, the creation of threads from sparks is entirely optional. This fact can be used to throttle the creation of threads from sparks in order to avoid swamping the parallel machine. This is a *lazy thread creation* approach (the term "lazy task creation" was coined later by Mohr et al. [7] to describe a similar mechanism in MultiLisp). A corollary is that, since sparks do not carry any execution state, they can be very lightweight. Generally, a single pointer is adequate to record a sparked expression in most implementations of GpH.

Sparks may be chosen for conversion to threads using a number of different strategies. One common approach is to use the oldest spark first. If the application is a divide-and-conquer program, this means that the spark representing the largest amount of work will be chosen. Alternatively, an approach may be used where the youngest (smallest) spark is executed locally and the oldest (largest) spark is offloaded for remote execution. This will improve locality, but may increase thread creation costs, since many of the locally created threads might otherwise be subsumed into their parent thread.

Parallel Graph Reduction

A second key issue that must be dealt with is that of thread synchronization. Efficient sequential non-strict functional language implementations generally use an evaluation technique called *graph reduction*, where a program builds a graph data structure representing the work that is needed to give the result of a program and gradually rewrites this using the functional rules defined in the program until the graph is sufficiently complete to yield the required result. This rewriting process is known as *reducing* the graph to some *normal form* (in fact *weak head normal form*). Each node in the graph represents an expression in the original program. Initially, this will be a single unevaluated expression corresponding to the result of the program (a "thunk"). As execution proceeds, thunks are evaluated, and each graph node is overwritten with its result. In this way, results are automatically shared between several consumers. Only graph nodes that actually contribute to the final result need to be rewritten. This means that unnecessary work can be avoided, a process that, in the sequential world, allows *lazy evaluation*.

The same mechanism naturally lends itself to parallel evaluation. Each shared node in the graph represents a possible synchronization point between two parallel threads. The first thread to evaluate a graph node will lock it. Any thread that evaluates the node in parallel with the thread that is evaluating it will then block when it attempts to read the value of the node. When the evaluating thread produces the result, the graph node will be updated and any blocked threads will be awoken. In this way, threads will automatically synchronize through the graph representing the computation, not just at the root of each thread, but whenever they share any sub-expressions.

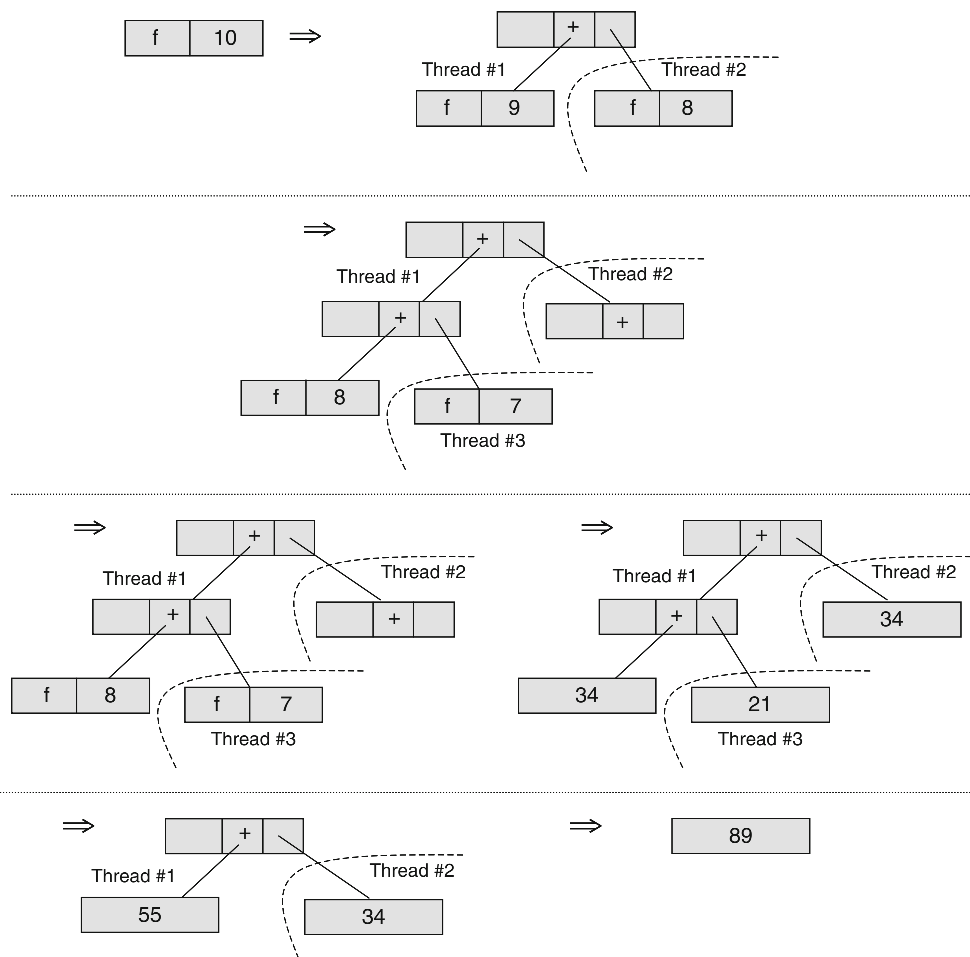
Figure 1 shows a simple example of a divide-and-conquer parallel program, where the root of the computation, `f 9`, is rewritten using three threads: one for the main computation and two sub-threads, one to evaluate `f 8` and one to evaluate `f 7`. These thunks are linked into the addition nodes in the main computation. Having evaluated the sparked thunks, the second and third threads update their root nodes with the corresponding result. Once the main thread has evaluated the remaining thunk (another call to `f 8`, which we have assumed is not shared with Thread #2), it will incorporate these results into its own result. The final stage of the computation is to rewrite the root of the graph (the result of the program) with the value `89`.

Evaluate-and-Die

As a thread evaluates its program graph, it may encounter a node that has been sparked. There are three possible cases, depending on the evaluation status of the sparked node. If the thunk has already been evaluated, then its value can be used as normal. If the thunk has not yet been evaluated, then the thread will evaluate it as normal, first annotating it to indicate that it is under evaluation. Once a result is produced, the node will be overwritten with the result value, and any blocked threads will be reawakened. Finally, if the node is actually under evaluation, then the thread must be blocked until the result is produced and the node updated with this value. This is the *evaluate-and-die* execution model [8]. The advantage of this approach is that it automatically absorbs sparks into already executing threads, so increasing their granularity and avoiding thread creation overheads. If a spark refers to a thunk that has already been evaluated then it may be discarded without a thread ever being created.

The *seq* Primitive

While *par* is entirely adequate for identifying many forms of parallelism, Roe and Peyton Jones discovered [13] that by adding a sequential combining function, called *seq*, much tighter control could be obtained over parallel execution. This could be used both to reduce the need for detailed knowledge of the underlying parallel implementation and to encode more sophisticated patterns of parallel execution. For example, one



Glasgow Parallel Haskell (GpH). Fig. 1 Parallel graph reduction

of the *par* constructs in the example above can be replaced by a *seq* construct, as shown below.

```
f x =
    let r1 = fibonacci x;
    r2 = factorial x in
    let result = (r1, r2) in
    r1 ‘par’ (r2 ‘seq’ result)
    -- was r1 ‘par’ r2 ‘par’ result
```

The *seq* construct acts like `;` in a conventional language such as C: it first evaluates its first argument (here `r2`), and then returns its second argument (here `result`). So in the example above, rather than creating two sparks as before, now only one is created. Previously, depending on the order in which threads were scheduled, either the parent thread would have blocked when it evaluated `r1` or `r2`, because these were already under evaluation, or one or both of the sparked threads would have blocked, because they were being evaluated (or had been evaluated) by the parent thread. Now, however, the only possible synchronization is between the thread evaluating `r1` and the parent thread. Note that it is not possible to use either of the simpler forms of `par r1 (r1, r2)` or `par r2 (r1, r2)` to achieve the same effect, as might be expected. Because the implementation is free to evaluate the pair `(r1, r2)` in whichever order it prefers, there is a 50% probability that the spark will block on the parent thread.

This is not the only use of *seq*. For example, evaluation strategies (discussed below) also make heavy use of this primitive to give precise control over evaluation order. While not absolutely essential for parallel evaluation, it is thus very useful to provide a finer

degree of control than can be achieved using *seq* alone. However, there are two major costs to the use of *seq*. The first is that the strictness properties of the program may be changed – this means that some thunks may be evaluated that were previously not needed, and that the termination properties of the program may therefore be changed. The second is that parallel programs may become *speculative*.

Speculative Evaluation

As described above, in GpH it is safe to *spark* any sub-expression that is needed by the result of the parallel computation. However, nothing prevents the programmer from sparking a sub-expression that is not known to be needed. This therefore allows speculative evaluation. Although there was some early experimentation with mechanisms to control speculation, these were found to be very difficult to implement, and current implementations of GpH do not provide facilities to kill created threads, change priorities dynamically, etc., as is necessary to support safe speculation. For example,

```
spec x y = x `par` y
```

defines a new function *spec* that sparks *x* and continues execution of *y*. If *x* is not definitely needed by *y*, or is a completely independent expression, this will spark *x* speculatively. If a thread is created to evaluate *x*, it will terminate either when it has completed evaluation of *x*, or when the main program terminates, having produced its own result, or if it evaluates an undefined value. In the latter case, it may cause the entire program to fail. Moreover, it is theoretically possible to prevent any progress on the main computation by flooding the runtime system with useless speculative threads. For these reasons, any speculative evaluation has to be treated carefully: if the program is to terminate, speculative sub-expressions should terminate in finite time without an error, and there should not be too many speculative sparks.

Evaluation Strategies

As discussed above, it is possible to construct higher-level parallel abstractions from basic *seq*/*par* constructs using standard Haskell programming constructs. These parallel abstractions can be associated with computation functions using higher-order definitions. This is the *evaluation strategy* approach. The key idea of evaluation strategies to separate what is to be evaluated from how it could be evaluated in parallel [14]. For example, a simple sequential ray tracing function could be defined as follows:

```
raytracer :: Int -> Int -> Scene ->
[Light] -> [[Vector]]
raytracer xlim ylim scene lights =
map traceline [0..ylim-1]
  where traceline y = [tracepixel scene
  lights x y | x <- [0..xlim-1]]
```

Given a visible image with maximum `x` and `y` dimensions of `xlim` and `ylim`, the `raytracer` function applies the `traceline` function to each line in the image (by mapping it across each value of `y` in the range `0..ylim-1`), and hence applies the `tracepixel` function to each pixel using the specified scene and lighting model. The `raytracer` function may be parallelized, for example, by adding the `parMap` strategy to parallelize each line as follows:

```
raytracer :: Int -> Int -> Scene ->
[Light] -> [[Vector]]
raytracer xlim ylim scene lights = map
traceline [0..ylim-1] `using` parMap rnf
```

Here, `parMap` is a strategy that specifies that each element of its value argument should be evaluated in parallel. It is parameterized on another strategy that specifies what to do with each element. Here, `rnf` indicates that each element should be evaluated as far as possible (`rnf` stands for "reduce to normal form"). The `using` function simply applies its second argument (an evaluation strategy) to its first argument (a functional value), and returns the value. It can be easily defined using higher-order functions and the *seq* primitive as follows:

```
using :: a -> Strategy a -> a
x `using` s = s x `seq` x
```

So, when a strategy is applied to an expression by the `using` operation, the strategy is first applied to the value of the expression, and once this has completed, the value is returned. This allows the use of both parallel and sequential strategies.

In fact, any list strategy may be used to parallelize the `raytracer` function instead of `parMap`. For example, a task farm could be used, or the list could be divided into equally sized chunks as shown below.

```
raytracer :: Int -> Int -> Scene ->
[Light] -> [[Vector]]
raytracer xlim ylim scene lights = ...
'using' parListChunk chunkSize rnf
```

Evaluation strategies have proved to be very powerful and flexible, having been successfully applied to several large programs, including symbolic programs with irregular patterns of parallelism [4]. Typically, only a few lines need to be changed at key points by adding appropriate `using` clauses. In abstracting parallel patterns from sequential computations, evaluation strategies have some similarities with *algorithmic skeletons*. The key differences between evaluation strategies and typical skeleton approaches is that evaluation strategies do not mandate a specific parallel implementation (since they rely on semi-explicit parallelism, they provide hints to the runtime system rather than directives); and that they are completely user-programmable – everything above the *par* and *seq* primitives is programmed using standard Haskell code. Unlike most skeleton approaches, they may also be easily composed to give irregular nested parallel structures or phased parallel computations, for example.

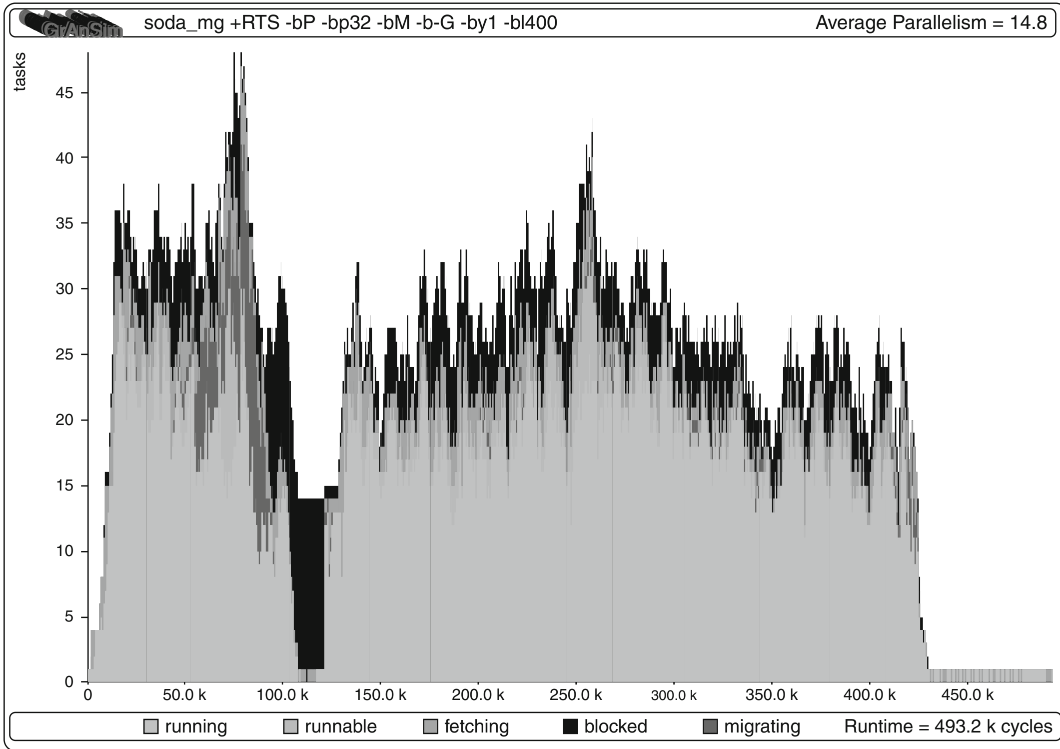
The GUM Implementation of GpH

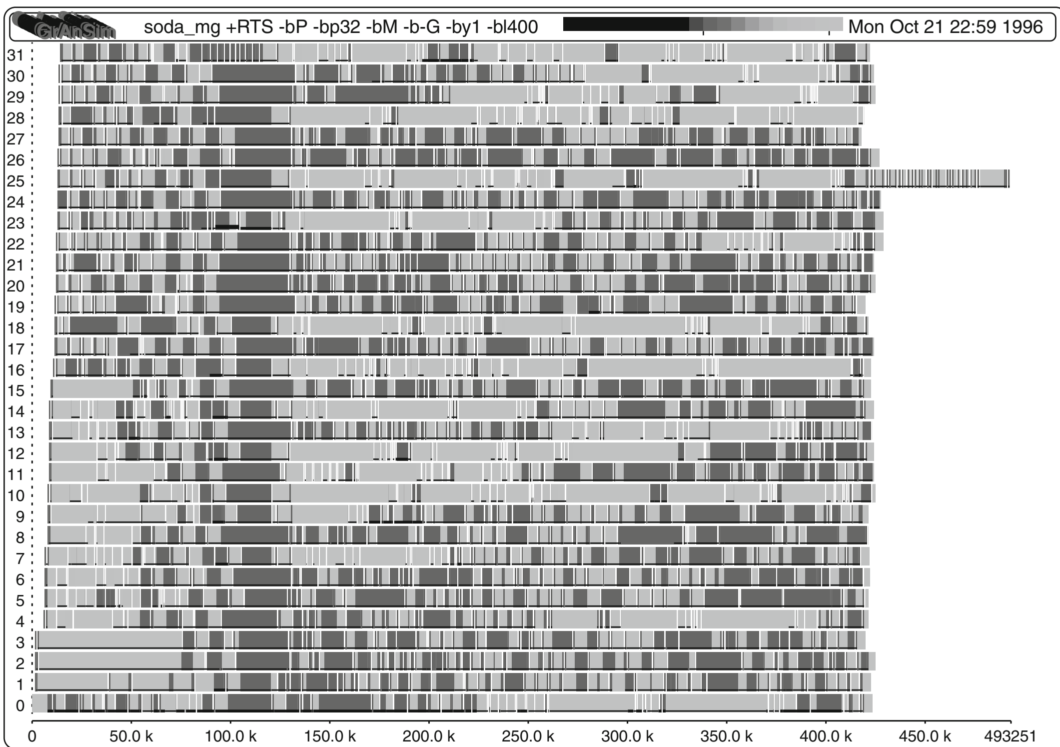
GUM is the original and most general implementation of GpH. It provides a virtual shared-memory abstraction for parallel graph reduction, using a message-passing implementation to target both physically shared-memory and distributed-memory systems. In the GUM model, the graph representing the program that is being evaluated is distributed among the set of processor elements (PEs) that are available to execute the program. These PEs usually correspond to the cores or processors that are available to execute the parallel program, but it is possible to map PEs onto multiple processes on the same core/processor, if required. Each PE manages its own execution environment, with its own local pools of sparks and threads, which it schedules as required. Sparks and threads are offloaded on demand to maintain good work balance.

A key feature of GUM is that it uses a two-level memory model: each PE has its own private heap that contains unshared program graph created by local threads. Within this heap, some graph nodes may be shared with other PEs. These nodes are given *global addresses*, which identify the owner of the graph node. The advantage of this model is that it allows completely independent memory management: by keeping tables of global in-pointers, which are used as garbage collection roots into the local heap, each PE can garbage-collect its own local heap independently of all other PEs. This local garbage collection is conservative, as required, since even if a global in-pointer is no longer referenced by any other PE, it will still be treated as a garbage collection root. In order to collect global garbage, a separate scheme is used, based on distributed reference counting. Since the majority of the program graph never needs to be shared, this approach brings major efficiency gains over typical single-level memory management. In addition to the reduced need for synchronization during garbage collection, there is also no need to maintain global locks across purely local graph. Within each PE, GUM uses the same efficient (and sequential) garbage collector as the standard GHC implementation. This is currently a stop-and-copy generational collector based on Appel's collector.

Visualizing the Behavior of GpH Programs

Good visualization is an essential part of understanding parallel behavior. Runtime information can be visualized at a variety of levels to give progressively more detailed information. For example, Fig. 2 visualizes the overall and per-PE parallel activity for a 32-PE system running the `soda` application (a simple crossword-puzzle solver). Apart from the clear phase-transition about 25% into the execution, very few threads are blocked; and there are very few runnable, but not running threads. Those that are runnable are migrated to balance overall system load. The profile also reveals that relatively little time is spent fetching nonlocal data. The per-PE profile gives a more detailed view. It is obvious that the workload is well balanced and that the work-stealing mechanism is effective in distributing work. The example also shows where PEs are idle for repeated periods, and that only the main PE is active at the end of the computation (collating results to be written as the output of the program). It thus clearly identifies places where improvements could be made to the parallel program.





Glasgow Parallel Haskell (GpH). Fig. 2 Sample overall and per-PE Activity Profiles for a 32-PE machine

G

The GHC-SMP Implementation of GpH

A recent development is the GHC-SMP [6] implementation of GpH from Microsoft Research Labs, Cambridge, UK, which is integrated into the standard GHC distribution. This provides an implementation of **GpH** that specifically targets shared-memory and multicore systems. It uses a similar model of PEs and threads to the **GUM** implementation, including spark pools and runnable thread pools, and implements a similar spark-stealing model. The key difference from GUM is that GHC-SMP uses a physically shared heap, rather than a virtual shared heap with an underlying message-passing implementation. This heap is garbage-collected using a global stop-and-copy collector that requires the synchronization of all PEs, but which may itself be executed in parallel using the available processor cores. Also, unlike GUM, GHC-SMP exports threads to remote PEs based on the load of the local core, rather than on demand.

GRID-GUM and the SymGrid System

By replacing the low-level communications library with MPICH-G2, it is possible to execute GUM (or Eden – see below) not only on standard clusters of workstations, but also on wide-area computational grids, coordinated by the standard Globus grid middleware. The GRID-GUM and grid-enabled Eden implementations form the basis for the SymGrid-Par middleware [3], which aims to provide high-level support for computational grids for a variety of symbolic computing systems. The middleware coordinates symbolic computing engines into a coherent parallel system, providing high-level skeletons. The system uses a high-level data-exchange protocol (SCSCP) based on the standard OpenMath XML format for mathematical data. Results have been very promising to date, with superlinear performance being achievable for some mathematical applications, without changing any of the sequential symbolic computing engines.

The GranSim Simulator

The GranSim parallel simulator was developed in 1995 [1] as an efficient and accurate simulator for GpH running on a sequential machine, specifically to expose granularity issues. It is unusual in modifying the sequential GHC runtime system so that evaluating graph nodes also triggers the addition of sparks, and simulates inter-PE communication. It thus follows the actual GUM implementation very precisely. It allows a range of communication costs to be simulated for a specific parallel application, ranging from zero (ideal communication) to costs that are similar to those Program execution times are also simulated, using a cost model that takes architectural characteristics into account. One of the key uses of GranSim is as the core of a parallel program development methodology, where a GpH program is first simulated under ideal parallel conditions, and then under communication cost settings for specific parallel machines: shared-memory, distributed-memory, etc. This allows the program to be gradually tuned for a specific parallel setting without needing access to the actual parallel machine, and in a way that is repeatable, can be easily debugged, and which provides detailed metrics.

GdH and Mobile Haskell

GdH [11] is a distributed Haskell dialect that builds on GpH. It adds explicit constructs for task creation on specific processors, with inter-processor communication through explicit mutable shared variables, that can be written on one processor and read on another. Internally, each processor runs a number of threads. The **GdH** implementation extends the GUM implementation with a number of explicit constructs. The main construct is `revalIO`, which constructs a new task on a specific procesor. In conjunction with mutable variables, this can be used to construct higher-level constructs. For example, the `showSystem` definition below prints the names of all available PEs. The `getAllHostNames` function maps the `hostName` function over the list of all processor identifiers, returning an IO action that obtains the environment variable `HOST` on the specified PE. This is used in the `showSystem` IO operation which first obtains a list of all host names, then outputs them as a sorted list, with duplicates eliminated using the standard `nub` function.

```
getAllHostNames :: IO [String]
getAllHostNames = mapM hostName allPEId

hostName :: PEId -> IO String
hostName pe = revalIO (getEnv ‘‘HOST’’) pe

showSystem = do { hostnames <=
```

```
getAllHostNames; showHostNames hostnames}
  where
     showHostNames names = putStrLn
     (show (sort (nub names)))
```

At the language level, GdH is similar to Concurrent Haskell, which is designed to execute explicit concurrent threads on a single PE. The key language difference is the inclusion of an explicit `PEId` to allocate tasks to PEs, supported by a distributed runtime environment. GdH is also broadly similar to the industrial Erlang language, targeting a similar area of distributed programming, but is non-strict rather than strict and, since it is a research language, does not have the range of telecommunications-specific libraries that Erlang supports in the form of the **Erlang**/OTP development platform. Mobile Haskell [12] similarly extends Concurrent Haskell, adding higher-order communication channels (known as Mobile Channels), to support mobile computations across dynamically evolving distributed systems. The system uses a bytecode implementation to ensure portability, and serializes arbitrary Haskell values including (higher-order) functions, IO actions, and even mobile channels.

Eden

Eden [5] (described in a separate encyclopedia entry) is closely related to GpH. Like GpH, it uses an implicit communication mechanism, sharing values through program graph nodes. However, it uses an explict `process` construct to create new processes and explicit process application to pass streams of values to each process. Unlike GpH, all data that is passed to an Eden process must be evaluated before it is communicated. There is also no automatic work-stealing mechanism, task migration, granularity agglomeration, or virtual shared graph mechanism. Eden, does, however, provide additional threading support: each output from the process is evaluated using its own parallel thread. It also provides mechanisms to allow explicit communication channels to be passed as first-class objects between processes.

Eden has been used to implement both algorithmic skeletons and evaluation strategies, where it has the advantage of providing more controllable parallelism but the disadvantage of losing adaptivity. One particularly useful technique is to use Eden to implement a master–worker evaluation strategy, where a set of dynamically generated worker functions is allocated to a fixed set of worker processors using an explicitly programmed scheduler. This allows Eden to deal with varying thread granularities, without using a lazy thread creation mechanism, as in GpH.

While it does not need the advanced adaptivity mechanisms that GUM provides, the Eden implementation shares several basic components, in particular, the communication library and scheduler have very similar implementations.

Current Status

GpH has been adopted as a significant part of the Haskell community effort on parallelism. The evaluation strategies library has just been released as part of the mainstream GHC compiler release, building on the GHC-SMP implementation, and there has also been significant recent effort on visualization with the release of the EdenTV and ThreadScope visualizers for Eden and GHC-SMP, respectively. Work is currently underway to integrate GUM and GHC-SMP to give a wide-spectrum implementatation for GpH, and Eden will also form part of this effort. The SymGrid-Par system is being developed as part of a major UK project to provide support for high-performance computing on massively parallel machines, such as HECToR.

Future Directions

The advent of multicore computers and the promise of manycore computers have changed the nature of parallel computing. The GpH model of lightweight multithreaded parallelism is well suited to this new order, offering the ability to easily generate large amounts of parallelism. The adaptivity mechanisms built into the GUM implementation mean that the parallel program can dynamically, and automatically, change its behavior to increase parallelism, improve locality, or throttle back parallelism as required.

Future parallel systems are likely to be built more hierarchically than present ones, with processors built from heteregeneous combinations of general-purpose cores, graphics processing units, and other specialist units, using several communication networks and a complex memory hierarchy. Proponents of manycore architectures anticipate that a single processor

will involve hundreds or even thousands of such units. Because of the cost of maintaining a uniform memory model across large numbers of systems, when deployed on a large scale these processors are likely to be combined hierarchically into multiple levels of clusters. These systems may then be used to form high performance "clouds." Future parallel languages and implementations must therefore be highly flexible and adaptable, capable of dealing with multiple levels of communication latency and internal parallel structure, and perhaps fault tolerance. The GpH model with evaluation strategies, supported by adaptive implementations such as GUM forms a good basis for this, but it will be necessary to extend the existing models and implementations to cover more heterogeneous processor types, to deal with multiple levels of parallelism and additional program structure, and to focus more directly on locality issues.

Related Entries

►Eden
►Fortress (Sun HPCS Language)
►Functional Languages
►Futures
►MPI (Message Passing Interface)
►MultiLisp
►NESL
►Parallel Skeletons
►Processes, Tasks, and Threads
►Profiling
►PVM (Parallel Virtual Machine)
►Shared-Memory Multiprocessors
►Sisal
►Speculation, Thread-Level

Bibliographic Notes and Further Reading

The main venues for publication on GpH and other parallel functional languages are the International Conference on Functional Programming (ICFP) and its satellite events including the Haskell Symposium; the International Symposium on Implementation and Application of Functional Languages (IFL); the International Conference on Programming Language Design and Implementation (PLDI); and the Symposium on Trends in Functional Programming (TFP). Papers also frequently appear in the Journal of Functional Programming (JFP). A survey of Haskell-based parallel languages and implementations, as of 2002, can be found in [16], and a general introduction to research in parallel functional programming, as of 1998, can be found in [2]. Some examples of the use of GpH in larger applications can be found in [4]. Much of this entry is based on private notes, e-mails, and final reports on the various research projects that have used GpH. The main paper on evaluation strategies is [14]. The main paper on the GUM implementation is [15], and the main paper on the GranSim similator is [1]. Many subsequent papers have used these systems and ideas. For example, one recent paper describes the SymGrid-Par system [3]. Further material may be found on the GpH web page at http://www.macs.hw.ac.uk/~dsg/gph/, on the GdH web page at http://www.macs.hw.ac.uk/~dsg/gdh/, on the GHC web page at http://www.haskell.org/ghc, on the SCIEnce project web page at http://www.symbolic-computation.org, on the Eden web page at http://www.mathematik.uni-marburg.de/~eden, and on the contributor's web page at http://www-fp.cs.st-andrews.ac.uk/~kh.

Bibliography

1. Hammond K, Loidl H-W, Partridge A (1995) Visualising granularity in parallel programs: a graphical winnowing system for Haskell. Proceedings of the HPFC'95 – Conference on High Performance Functional Computing, Denver, pp 208–221, 10–12 April 1995
2. Hammond K, Michaelson G (eds) (1999) Research directions in parallel functional programming. Springer, Heidelberg
3. Hammond K, Zain AA, Cooperman G, Petcu D, Trinder PW (2007) SymGrid: a framework for symbolic computation on the grid. Proceedings of the EuroPar '07: 13th International EuroPar Conference, Springer LNCS 4641, Rennes, France, pp 457–466
4. Loidl H-W, Trinder P, Hammond K, Junaidu SB, Morgan RG, Peyton Jones SL (1999) Engineering parallel symbolic programs in GpH. Concurrency Pract Exp 11(12):701–752
5. Loogen R, Ortega-Mallén Y, Peña Mar R (2005) Parallel functional programming in Eden. J Functional Prog 15(3):431–475
6. Marlow S, Jones SP, Singh S (2009) Runtime support for multicore Haskell. Proceedings of the ICFP '09: 14th ACM SIGPLAN International Conference on Functional Programming, ACM Press, New York
7. Mohr E, Kranz DA, Halstead RH Jr (1991) Lazy task creation: a technique for increasing the granularity of parallel programs. IEEE Trans Parallel Distrib Syst 2(3):264–280
8. Peyton Jones S, Clack C, Salkild J (1989) High-performance parallel graph reduction. Proceedings of the PARLE'89 – Conference

on Parallel Architectures and Languages Europe, Springer LNCS 365, pp 193–206
9. Peyton Jones S, Hall C, Hammond K, Partain W, Wadler P (1993) The Glasgow Haskell compiler: a technical overview. Proceedings of the JFIT (Joint Framework for Information Technology) Technical Conference, Keele, UK, pp 249–257
10. Peyton Jones S (ed), Hughes J, Augustsson L, Barton D, Boutel B, Burton W, Fasel J, Hammond K, Hinze R, Hudak P, Johnsson T, Jones M, Launchbury J, Meijer E, Peterson J, Reid A, Runciman C, Wadler P (2003) Haskell 98 language and libraries. The revised report. Cambridge University Press, Cambridge
11. Pointon R, Trinder P, Loidl H-W (2000) The design and implementation of Glasgow distributed Haskell. Proceedings of the IFL'00 – 12th International Workshop on the Implementation of Functional Languages, Springer LNCS 2011, Aachen, Germany, pp 53–70
12. Rauber Du Bois A, Trinder P, Loidl H (2005) mHaskell: mobile computation in a purely functional language. J Univ Computer Sci 11(7):1234–1254
13. Roe P (1991) Parallel programming using functional languages. PhD thesis, Department of Computing Science, University of Glasgow
14. Trinder P, Hammond K, Loidl H-W, Peyton Jones S (1998) Algorithm + strategy = parallelism. J Funct Prog 8(1):23–60
15. Trinder P, Hammond K, Mattson J Jr, Partridge A, Peyton Jones S (1996) GUM: a portable parallel implementation of Haskell. Proceedings of the PLDI'96 – ACM Conference on Programming Language Design and Implementation, Philadelphia, pp 79–88
16. Trinder P, Loidl H-W, Pointon R (2002) Parallel and distributed Haskells. J Funct Prog 12(4&5):469–510. Special Issue on Haskell

Global Arrays

►Global Arrays Parallel Programming Toolkit

Global Arrays Parallel Programming Toolkit

Jarek Nieplocha[1,†], Manojkumar Krishnan[1],
Bruce Palmer[1], Vinod Tipparaju[2],
Robert Harrison[2], Daniel Chavarría-Miranda[1]
[1]Pacific Northwest National Laboratory, Richland, WA, USA
[2]Oak Ridge National Laboratory, Oak Ridge, TN, USA

Synonyms

Global arrays; GA

Definition

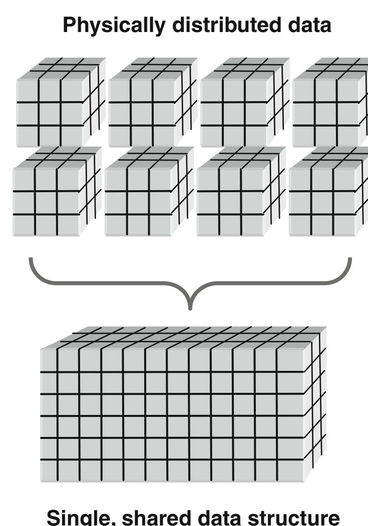
Global Arrays is a high-performance programming model for scalable, distributed-memory, parallel computer systems. Global Arrays is based on the concept of globally accessible dense arrays that are logically shared, yet physically distributed onto the memories of a parallel distributed computer system (Fig. 1 illustrates this concept).

Discussion

Introduction

Global Arrays (GA) is a high-performance programming model for scalable, distributed-memory, parallel computer systems. GA is a library-based Partitioned Global Address Space (PGAS) programming model. The underlying supported sequential languages are Fortran, C, C++, and Python. GA provides *global view access* to very large dense arrays through API functions implemented for those languages, under a Single Program Multiple Data (SPMD) execution environment.

GA was originally developed as part of the underlying software infrastructure for the US Department of Energy's NWChem computational chemistry software package. Over time, it has been developed into a standalone package with a rich set of API functions (200+) that cater to many needs in scientific application development. GA has been used to enable scalable



Global Arrays Parallel Programming Toolkit. Fig. 1 Dual view of Global Arrays data structures

† deceased.

parallel execution for several major scientific applications including NWChem (computational chemistry, specifically electronic structure calculation), STOMP (Subsurface Transport Over Multiple Phases, a subsurface flow and transport simulator), ScalaBLAST (a more scalable, higher-performance version of BLAST), Molpro (quantum chemistry), TE^2THYS (unstructured, implicit CFD and coupled fluid/solid mechanics finite volume code), Pagoda (Parallel Analysis of Geodesic Data), COLUMBUS (computational chemistry), and GAMESS-UK (computational chemistry).

GA's development has occurred over the last two decades. For this reason, the number of people involved and their contributions is large. GA's original development occurred as a co-design effort between the NWChem team and the computer science team focused on GA. The main designer and original developer of GA was Jarek Nieplocha. Robert Harrison led the main effort in the development of NWChem.

Basic Global Arrays

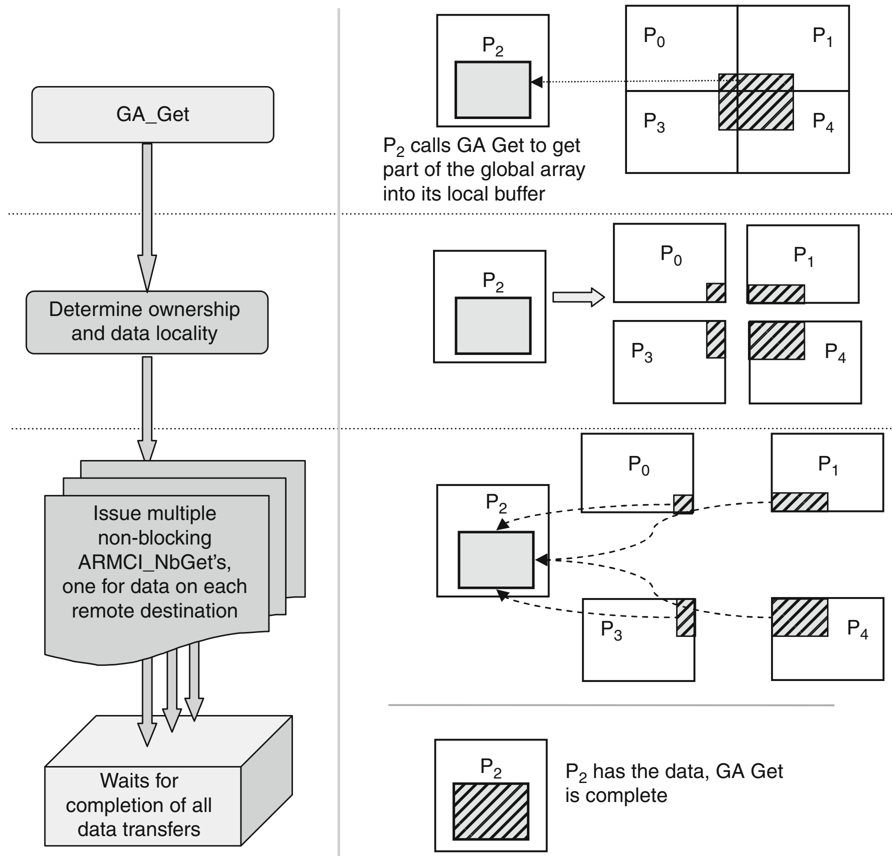
There are three classes of operations in Global Arrays: core operations, task parallel operations, and data-parallel operations. These operations have multiple language bindings, but provide the same functionality independent of the language. The current GA library contains approximately 200 operations that provide a rich set of functionality related to data management and computations involving distributed arrays. GA is interoperable with MPI, enabling the development of hybrid programs that use both programming models.

The basic components of the Global Arrays toolkit are function calls to create global arrays, copy data to, from, and between global arrays, and identify and access the portions of the global array data that are held locally. There are also functions to destroy arrays and free up the memory originally allocated to them. The basic function call for creating new global arrays is **`nga_create`**. The arguments to this function include the dimension of the array, the number of indices along each of the coordinate axes, and the type of data (integer, float, double, etc.) that each array element represents. The function returns an integer handle that can be used to reference the array in all subsequent operations. The allocation of data can be left completely to the toolkit, but if it is desirable to control the distribution of data for load balancing or other reasons, additional versions of the **`nga_create`** function are available that allow the user to specify in detail how data is distributed between processors. The basic **`nga_create`** call provides a simple mechanism to control data distribution via the specification of an array that indicates the minimum dimensions of a block of data on each processor.

One of the most important features of Global Arrays is the ability to easily move blocks of data between global arrays and local buffers. The data in the global array can be referred to using a global indexing scheme and data can be moved in a single function call, even if it represents data distributed over several processors. The **`nga_get`** function can be used to move a block of distributed data from a global array to a local buffer. The arguments consist of the array handle for the array that data is being taken from, two integer arrays representing the lower and upper indices that bound the block of distributed data that is going to be moved, a pointer to the local buffer or a location in the local buffer that is to receive the data, and an array of strides for the local data. The **`nga_put`** call is similar and can be used to move data in the opposite direction.

The number of basic GA operations is fairly small and many parallel programs can be written with just the following ten routines:

- `GA_Initialize()`: Initialize the GA library.
- `GA_Terminate()`: Release internal resources and finalize execution of a GA program.
- `GA_Nnodes()`: Return the number of GA compute processes (corresponds to the SPMD execution environment).
- `GA_Nodeid()`: Return the GA process ID of the calling compute process, this is a number between 0 and `GA_Nnodes() - 1`.
- `NGA_Create()`: Create an *n*-dimensional globally accessible dense array (global array instance).
- `NGA_Destroy()`: Deallocate memory and resources associated with a global array instance.
- `NGA_Put()`: Copy data from a local buffer to an array section within a global array instance in a one-sided manner.
- `NGA_Get()`: Copy data from an array section within a global array instance to a local buffer in a



Global Arrays Parallel Programming Toolkit. Fig. 2 *Left: GA_Get* flow chart. *Right*: An example: Process *P2* issues *GA_Get* to get a chunk of data, which is distributed (partially) among *P0, P1, P3*, and *P4* (owners of the chunk)

one-sided manner (see Fig. 2 for a detailed description of how get() operates).

- `GA_Sync()`: Synchronize compute processes via a barrier and ensure that all pending GA operations are complete (in accordance to the GA *consistency model*).
- `NGA_Distribution()`: Returns the array section owned by a specified compute process.

Example GA Program

We present a parallel matrix multiplication program written in Global Arrays using the Fortran language interface. It uses most of the basic GA calls described before, in addition to some more advanced calls to create global array instances with specified data distributions. The program computes the result of $C = A \times B$. Some variable declarations have been omitted for brevity.

Discussion on the Example Program

The program is (mostly) a fully functional GA code, except for omitted variable declarations. It creates global array instances with specific data distributions, illustrates the use of the `nga_put()` and `nga_get()` primitives, as well as locality information through the `nga_distribution()` call. The code includes calls to initialize and terminate the MPI library, which are needed to provide the SPMD execution environment to the GA application. (It is possible to write a GA application that does not call the MPI library through the use of the TCGMSG simple message-passing environment included with GA.) The code includes the creation and use of local buffers to be used as sources and targets for put and get operations (`lA, lB, lC`), which in this case were allocated as Fortran 90 dynamic arrays.

Lines 43–69 contain the principal part of the example code and illustrate several of the features of

```
 1 program matmul
 2 integer :: sz
 3 integer :: i, j, k, pos, g_a, g_b, g_c
 4 integer :: nproc, me, ierr
 5 integer, dimension(2) :: dims, nblock, chunks
 6 integer, dimension(1) :: lead
 7 double precision, dimension(:,:), pointer :: lA, lB
 8 double precision, dimension(:,:), pointer :: lC
 9 call mpi_init(ierr)

10 call ga_initialize()
11 nproc = ga_nnodes()
12 me = ga_nodeid()

13 if (me .eq. 0) then
14   write (*, *) 'Running on: ', nproc, ' processors'
15 end if

16 dims(:) = sz
17 chunks(:) = sz/sqrt(dble(nproc)) ! only runs on a perfect square number
   of processors
18 nblock(1) = sz/chunks(1)
19 nblock(2) = sz/chunks(2)
20 allocate(dmap(nblock(1) + nblock(2)))

21 pos = 1
22 do i = 1, sz - 1, chunks(1) ! compute beginning coordinate of each
   partition in the 1st dimension
23   dmap(pos) = i
24   pos = pos + 1
25 end do

26 do j = 1, sz - 1, chunks(2) ! compute beginning coordinate of each
   partition in the 2nd dimension
27   dmap(pos) = j
28   pos = pos + 1
29 end do

30 ret = nga_create_irreg(MT_DBL, ubound(dims), dims, 'A', dmap, nblock, g_a) !
   create a global array instance with specified data distribution
31 ret = ga_duplicate(g_a, g_b, 'B') ! duplicate same data distribution for
   array B
32 ret = ga_duplicate(g_a, g_c, 'C') ! and C

33 allocate(lA(chunks(1), chunks(2)), lB(chunks(1), chunks(2)),
   lC(chunks(1), chunks(2)))
34 lA(:, :) = 1.0
```

```
35 lB(:, :) = 2.0
36 lC(:, :) = 0.0
37 lead(1) = chunks(2)

38 call nga_distribution(g_a, me, tcoordsl, tcoordsh)

39 ! initialize global array instances to respective values
40 call nga_put(g_a, tcoordsl, tcoordsh, lA(1, 1), lead)
41 call nga_put(g_b, tcoordsl, tcoordsh, lB(1, 1), lead)
42 call nga_put(g_c, tcoordsl, tcoordsh, lC(1, 1), lead)

43 ! obtain all blocks in the row of the A matrix
44 tcoordsl1(1) = 1
45 tcoordsl1(2) = tcoordsl(2)
46 tcoordsh1(1) = chunks(1)
47 tcoordsh1(2) = tcoordsh(2)

48 ! obtain all blocks in the column of the B matrix
49 tcoordsl2(1) = tcoordsl(1)
50 tcoordsl2(2) = 1
51 tcoordsh2(1) = tcoordsh(1)
52 tcoordsh2(2) = chunks(2)

53 do pos = 1, nblock(1) ! matrix is square
54   call nga_get(g_a, tcoordsl1, tcoordsh1, lA(1, 1), lead)
55   call nga_get(g_b, tcoordsl2, tcoordsh2, lB(1, 1), lead)

56   do j = 1, n
57     do k = 1, n
58       do i = 1, n
59         lC(i, j) = lC(i, j) + lA(i, k) * lB(k, j)
60       end do
61     end do

62   ! advance coordinates for blocks
63   tcoordsl1(1) = tcoordsl1(1) + chunks(1)
64   tcoordsh1(1) = tcoordsh1(1) + chunks(1)

65   tcoordsl2(2) = tcoordsl2(2) + chunks(2)
66   tcoordsh2(2) = tcoordsh2(2) + chunks(2)
67 end do
68 ! lC contains the final result for the block owned by the process
69 call nga_put(g_c, tcoordsl, tcoordsh, lC(1, 1), lead)

70 ! do something with the result
71 call ga_print(g_c)
72 deallocate(dmap, lA, lB, lC)
```

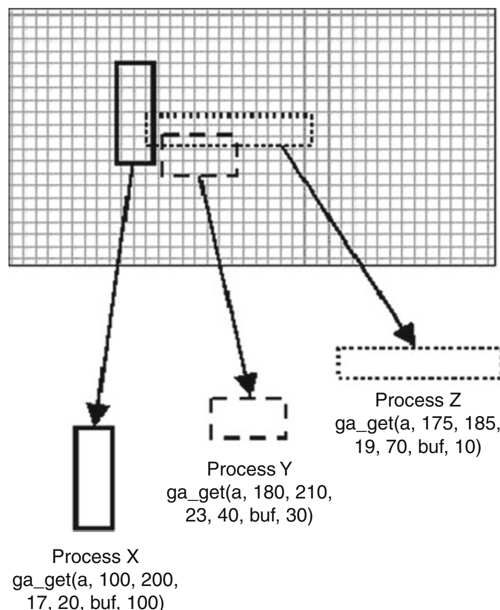
```
73 ret = ga_destroy(g_a)
74 ret = ga_destroy(g_b)
75 ret = ga_destroy(g_c)
76 call ga_terminate()
77 call mpi_finalize(ierr)

78 end program matmul
```

GA: data is being accessed using *global coordinates* (lines 54–55 and 63–66) that correspond to the global size of the global array instances that were created previously; in addition, the access to the global array data for arrays `g_a` and `g_b` in lines 54 and 55 is done *without requiring the participation* of the process where that data is allocated (one-sided access). Figure 3 illustrates the concept of one-sided access.

Global Arrays Concepts

GA allows the programmer to control data distribution and makes the locality information readily available to be exploited for performance optimization. For example, global arrays can be created by: (1) allowing the library to determine the array distribution, (2) specifying the decomposition for only one array dimension and allowing the library to determine the others, (3) specifying the distribution block size for all dimensions, or (4) specifying an irregular distribution as a Cartesian product of irregular distributions for each axis. The distribution and locality information is always available through interfaces that allow the application developer to query: (1) which data portion is held by a given process, (2) which process owns a particular array element, and (3) a list of processes and the blocks of data owned by each process corresponding to a given section of an array.



Global Arrays Parallel Programming Toolkit. Fig. 3 Any part of GA data can be accessed independently by any process at any time

The primary mechanisms provided by GA for accessing data are block copy operations that transfer data between layers of memory hierarchy, namely, global memory (distributed array) and local memory. Further, extending the benefits of using blocked data accesses and copying remote locations into contiguous local memory can improve cache performance by reducing both conflict and capacity misses [1]. In addition, each process is able to access directly the data held in a section of a Global Array that is locally assigned to that process. Data representing sections of the Global Array owned by other processes on SMP clusters can also be accessed directly using the GA interface, if desired. Atomic operations are provided that can be used to implement synchronization and assure correctness of an accumulate operation (floating-point sum reduction that combines local and remote data) executed concurrently by multiple processes and targeting overlapping array sections.

GA is extensible as well. New operations can be defined exploiting the low-level interfaces dealing with distribution, locality, and providing direct memory access (**`nga_distribution, nga_locate_region, nga_access, nga_release, nga_release_update`**). These, e.g., were used to provide

additional linear algebra capabilities by interfacing with third-party libraries, e.g., ScaLAPACK [2].

Global Arrays Memory Consistency Model

In shared-memory programming, one of the issues central to performance and scalability is memory consistency. Although the sequential consistency model [3] is straightforward to use, weaker consistency models [4] can offer higher performance on modern architectures and they have been implemented on actual hardware. GA's nature as a one-sided, global-view programming model requires similar attention to memory consistency issues. The GA approach is to use a weaker-than-sequential consistency model that is still relatively straightforward to understand by an application programmer. The main characteristics of the GA approach include:

- GA distinguishes two types of completion of the store operations (i.e., put, scatter) targeting global shared memory: local and remote. The blocking store operation returns after the operation is completed locally, i.e., the user buffer containing the source of the data can be reused. The operation completes remotely after either a memory fence operation or a barrier synchronization is called. The fence operation is required in critical sections of the user code, if the globally visible data is modified.
- The blocking operations (get/put) are ordered only if they target overlapping sections of global arrays. Operations that do not overlap or access different arrays can complete in arbitrary order.
- The nonblocking get/put operations complete in arbitrary order. The programmer uses wait/test operations to order completion of these operations, if desired.

Global Arrays Extensions

To allow the user to exploit data locality, the toolkit provides functions identifying the data from the global array that is held locally on a given processor. Two functions are used to identify local data. The first is the **nga_distribution** function, which takes a processor ID and an array handle as its arguments and returns a set of lower and upper indices in the global address space representing the local data block. The second is the **nga_access** function, which returns an array index and an array of strides to the locally held data. In Fortran, this can be converted to an array by passing it through a subroutine call. The C interface provides a function call that directly returns a pointer to the local data.

In addition to the communication operations that support task parallelism, the GA toolkit includes a set of interfaces that operate on either entire arrays or sections of arrays in the data-parallel style. These are collective data-parallel operations that are called by all processes in the parallel job. For example, movement of data between different arrays can be accomplished using a single function call. The **nga_copy_patch** function can be used to move a patch, identified by a set of lower and upper indices in the global index space, from one global array to a patch located within another global array. The only constraints on the two patches are that they contain equal numbers of elements. In particular, the array distributions do not have to be identical, and the implementation can perform, as needed, the necessary data reorganization (the so-called MxN problem [5]). In addition, this interface supports an optional transpose operation for the transferred data. If the copy is from one patch to another on the same global array, there is an additional constraint that the patches do not overlap.

Historical Development and Comparison with Other Programming Models

The original GA package [6–8] offered basic one-sided communication operations, along with a limited set of collective operations on arrays in the style of BLAS [9]. Only two-dimensional arrays and two data types were supported. The underlying communication mechanisms were implemented on top of vendor-specific interfaces. In the course of 10 years, the package evolved substantially and the underlying code was completely rewritten. This included separation of the GA internal one-sided communication engine from the high-level data structure. A new portable, general, and GA-independent communication library called ARMCI was created [10]. New capabilities were later added to GA without the need to modify the ARMCI interfaces. The GA toolkit evolved in multiple directions:

- Adding support for a wide range of data types and virtually arbitrary array ranks.
- Adding advanced or specialized capabilities that address the needs of some new application areas,

e.g., ghost cells or operations for sparse data structures.

- Expansion and generalization of the existing basic functionality. For example, mutex and lock operations were added to better support the development of shared-memory-style application codes. They have proven useful for applications that perform complex transformations of shared data in task parallel algorithms, such as compressed data storage in the multireference configuration interaction calculation in the COLUMBUS package [11].
- Increased language interoperability and interfaces. In addition to the original Fortran interface, C, Python, and a C++ class library were developed.
- Developing additional interfaces to third-party libraries that expand the capabilities of GA, especially in the parallel linear algebra area: ScaLAPACK [2] and SUMMA [12]. Interfaces to the TAO optimization toolkit have also been developed [13].
- Developed support for multilevel parallelism based on processor groups in the context of a shared-memory programming model, as implemented in GA [14, 15].

These advances generalized the capabilities of the GA toolkit and expanded its appeal to a broader set of applications. At the same time, the programming model, with its emphasis on a shared-memory view of the data structures in the context of distributed memory systems with a hierarchical memory, is as relevant today as it was in 1993 when the project started.

Comparison with Other Programming Models

The two predominant classes of programming models for parallel computers are distributed-memory, shared-nothing, and Uniform Memory Access (UMA) shared-everything models. Both the shared-everything and fully distributed models have advantages and shortcomings. The UMA shared-memory model is easier to use but it ignores data locality/placement. Given the hierarchical nature of the memory subsystems in modern computers, this characteristic can have a negative impact on performance and scalability. Careful code restructuring to increase data reuse and replacing fine-grained load/stores with block access to shared data can address the problem and yield performance for shared memory that is competitive with message passing [16]. However, this performance comes at the cost of compromising the ease of use that the UMA shared-memory model posits. Distributed, shared-nothing memory models, such as message-passing or one-sided communication, offer performance and scalability but they are more difficult to program. The classic message-passing paradigm not only transfers data but also synchronizes the sender and receiver. Asynchronous (nonblocking) send/receive operations can be used to diffuse the synchronization point, but cooperation between sender and receiver is still required. The synchronization effect is beneficial in certain classes of algorithms, such as parallel linear algebra, where data transfer usually indicates completion of some computational phase; in these algorithms, the synchronizing messages can often carry both the results and a required dependency. For other algorithms, this synchronization can be unnecessary and undesirable, and a source of performance degradation and programming complexity.

The Global Arrays toolkit [6–8] attempts to offer the best features of both models. It implements a *global-view* programming model, based on one-sided communication, in which data locality is managed by the programmer. This management is achieved by calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to the distributed shared-memory models that provide, e.g., an explicit acquire/release protocol [17]. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be specified by the programmer and hence managed. GA is related to the global address space languages such as UPC [18], Titanium [19], and, to a lesser extent, Co-Array Fortran [20]. In addition, by providing a set of data-parallel operations, GA is also related to data-parallel languages such as HPF [21], ZPL [22], and Data Parallel C [23]. However, the Global Array programming model is implemented as a library that works with most languages used for technical computing and does not rely on compiler technology for achieving parallel efficiency. It also supports a combination of task and data parallelism and is fully interoperable with the message-passing (MPI) model. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems [24], and by recognizing the communication overhead for remote data

transfers, it promotes data reuse and locality of reference. Virtually all scalable architectures possess nonuniform memory access characteristics that reflect their multilevel memory hierarchies. These hierarchies typically comprise processor registers, multiple levels of cache, local memory, and remote memory. Over time, both the number of levels and the cost (in processor cycles) of accessing deeper levels have been increasing. Scalable programming models must address memory hierarchy since it is critical to the efficient execution of applications.

Related Entries

►Coarray Fortran
►MPI (Message Passing Interface)
►PGAS (Partitioned Global Address Space) Languages
►UPC

Bibliographic Notes and Further Reading

A more detailed version of this entry has been published in the *International Journal of High Performance Computing Applications*, vol. 20, no. 2, May 2006 by SAGE Publications, Inc., All rights reserved. © 2006.

Bibliography

1. Lam MS, Rothberg EE, Wolf ME (1991) Cache performance and optimizations of blocked algorithms. In: Proceedings of the 4th international conference on architectural support for programming languages and operating systems, Santa Clara, 8–11 Apr 1991
2. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK: a linear algebra library for message-passing computers. In: Proceedings of eighth SIAM conference on parallel processing for scientific computing, Minneapolis
3. Scheurich C, Dubois M (1987) Correct memory operation of cache-based multiprocessors. In: Proceedings of 14th annual international symposium on computer architecture, Pittsburgh
4. Dubois M, Scheurich C, Briggs F (1986) Memory access buffering in multiprocessors. In: Proceedings of 13th annual international symposium on Computer architecture, Tokyo, Japan
5. CCA-Forum. Common component architecture forum. http://www.cca-forum.org
6. Nieplocha J, Harrison RJ, Littlefield RJ (1994) Global arrays: a portable shared memory programming model for distributed memory computers. In: Proceedings of Supercomputing, Washington, DC, pp 340–349
7. Nieplocha J, Harrison RJ, Littlefield RJ (1996) Global arrays: A nonuniform memory access programming model for high-performance computers. J Supercomput 10:169–189
8. Nieplocha J, Harrison RJ, Krishnan M, Palmer B, Tipparaju V (2002) Combining shared and distributed memory models: Evolution and recent advancements of the Global Array Toolkit. In: Proceedings of POHLL' 2002 workshop of ICS-2002, New York
9. Dongarra JJ, Croz JD, Hammarling S, Duff I (1990) Set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16:1–17
10. Nieplocha J, Carpenter B (1999) ARMCI: a portable remote memory copy library for distributed array libraries and compiler runtime systems. In: Proceedings of RTSPP of IPPS/SDP'99, San Juan, Puerto Rico
11. Dachsel H, Nieplocha J, Harrison RJ (1998) An out-of-core implementation of the COLUMBUS massively-parallel multireference configuration interaction program. In: Proceedings of high performance networking and computing conference, SC'98, Orlando
12. VanDeGeijn RA, Watts J (1997) SUMMA: Scalable universal matrix multiplication algorithm. Concurr Pract Exp 9:255–274
13. Benson S, McInnes L, Moré JJ Toolkit for Advanced Optimization (TAO). http://www.mcs.anl.gov/tao
14. Nieplocha J, Krishnan M, Palmer B, Tipparaju V, Zhang Y (2005) Exploiting processor groups to extend scalability of the GA shared memory programming model. In: Proceedings of ACM computing frontiers, Italy
15. Krishnan M, Alexeev Y, Windus TL, Nieplocha J (2005) Multilevel parallelism in computational chemistry using common component architecture and global arrays. In: Proceedings of Supercomputing, Seattle
16. Shan H, Singh JP (2000) A comparison of three programming models for adaptive applications on the origin2000. In: Proceedings of supercomputing, Dallas
17. Zhou Y, Iftode L, Li K (1996) Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In: Proceedings of operating systems design and implementation symposium, Seattle, pp 75–88
18. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K (1999) Introduction to UPC and language specification. Center for Computing Sciences CCS-TR-99-157, IDA Center for Computing Sciences, Bowie
19. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998) Titanium: A high-performance Java dialect. Concurr Pract Exp 10:825–836
20. Numrich RW, Reid JK (1998) Co-array Fortran for parallel programming. ACM Fortran Forum 17:1–31
21. High Performance Fortran Forum (1993) High Performance Fortran Language Specification, version 1.0. Sci Program 2(1):1–170
22. Snyder L (1999) A programmer's guide to ZPL. MIT Press, Cambridge
23. Hatcher PJ, Quinn MJ (1991) Data-parallel programming on MIMD computers. MIT Press, Cambridge
24. Nieplocha J, Harrison RJ, Foster I (1996) Explicit management of memory hierarchy. Adv High Perform Comput 185–200

Gossiping

►Allgather

GpH (Glasgow Parallel Haskell)

►Glasgow Parallel Haskell (GpH)

GRAPE

Junichiro Makino
National Astronomical Observatory of Japan, Tokyo, Japan

Definition

GRAPE (GRAvity PipE) is the name of a series of special-purpose computers designed for the numerical simulation of gravitational many-body systems. Most of GRAPE machines consist of hardwired pipeline processors to calculate the gravitational interaction between particles and programmable computers to handle all other works. GRAPE-DR (Greatly Reduced Array of Processor Elements with Data Reduction) replaced the hardwired pipeline by simple SIMD programmable processors.

Discussion

Introduction

The improvement in the speed of computers has been a factor of 100 in every decade, for the last 60 years. In these 60 years, however, the computer architecture has become more and more complex. Pipelined architecture were introduced in 1960s, and vector architectures became the mainstream in 1970s. In 1980s, a number of parallel architectures appeared, but in the 1990s and 2000s, distributed memory parallel computers built from microprocessors have taken over.

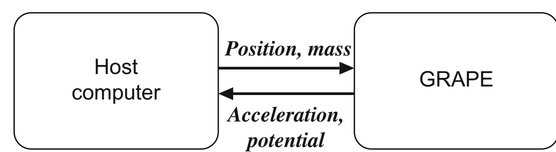
The technological driving force of this evolution of computer architecture has been the increase of the number of available transistors in integrated circuits, at least after the invention of integrated circuits in 1960s. In the case of CMOS LSIs, the number of transistors in a chip doubles in every 18 months. Under the assumption of so-called CMOS scaling, this means that the switching speed doubles in every 36 months. In the last 30 years, the number of transistors available on an LSI chip increased by roughly a factor of one million.

One way to make use of this huge number of transistors is to implement application-specific pipeline processors into a chip. The GRAPE series of special-purpose computers is one of such efforts to make efficient use of large number of transistors available on LSIs.

In many scientific simulations, it is necessary to solve N-body problems numerically. The gravitational N-body problem is one such example, which describes the evolution of many astronomical objects from the solar system to the entire universe. In some cases, it is important to treat non-gravitational effects such as the hydrodynamical interaction, radiation, and magnetic fields, but the gravity is the primary driving force that shapes the universe.

To solve the gravitational N-body problem, one needs to calculate the gravitational force on each body (particle) in the system from all other particles in the system. There are many ways to do so, and if relatively low accuracy is sufficient, one can use the Barnes–Hut tree algorithm [3] or FMM [5]. Even with these schemes, the calculation of the gravitational interaction between particles (or particles and multipole expansions of groups of particles) is the most time-consuming part of the calculation. Thus, one can greatly improve the speed of the entire simulation, just by accelerating the speed of the calculation of particle–particle interaction. This is the basic idea behind GRAPE computers.

The basic idea is shown in Fig. 1. The system consists of a host computer and special-purpose hardware, and the special-purpose hardware handles the calculation of gravitational interaction between particles. The host computer performs other calculations such as the time integration of particles, I/O, and diagnostics.



GRAPE. Fig. 1 Basic structure of a GRAPE system

This architecture accelerates not only the simple algorithm in which the force on a particle is calculated by taking the summation of forces from all other particles in the system, but also the Barnes–Hut tree algorithms and FMM. Moreover, it can be used with individual timestep algorithms [1], in which particles have their own times and timesteps and integrated in an event-driven fashion. The use of individual timestep is critical in simulations of many systems including star clusters and planetary systems, where close encounters and physical collisions of two particles require very small timesteps for a small number of particles.

History

The GRAPE project started in 1988. The first machine completed, the GRAPE-1 [7], was a single-board unit on which around 100 IC and LSI chips were mounted and wire-wrapped. The pipeline processor of GRAPE-1 was implemented using commercially available IC and LSI chips. It was a natural consequence of the fact that project members lacked both money and experience to design custom LSI chips. In fact, none of the original design and development team of GRAPE-1 had the knowledge of electronic circuit more than what was learned in basic undergraduate course for physics students.

For GRAPE-1, an unusually short word format was used, to make the hardware as simple as possible. The input coordinates are expressed in 15-bit fixed point format. After subtraction, the result is converted to 8-bit logarithmic format, in which 3 bit are used for the "fractional" part. This format is used for all following operations except for the final accumulation. The final accumulation was done in 48-bit fixed point, to avoid overflow and underflow. The advantage of the short word format is that ROM chips can be used to implement complex functions that require two inputs. Any function of two 8-bit words can be implemented by one ROM chip with 16-bit address input. Thus, all operations other than the initial subtraction of the coordinates and final accumulation of the force were implemented by ROM chips.

The use of extremely short word format in GRAPE-1 was based on the detailed theoretical analysis of error propagation and numerical experiment [13]. There are three dominant sources of error in numerical simulations of gravitational many-body systems. The first one is the error in the numerical integration of orbits of particles. The second one is the error in the calculated accelerations themselves. The third one comes from the fact that in many cases, the number of particles used is much smaller than the number of stars in the real systems such as a galaxy.

Whether or not the third one should be regarded as the source of error depends on the problem one wants to study. If the problem is, for example, merging of two galaxies, which takes place in relatively short timescale (compared to the orbital period of typical stars in galaxies), the effect of small number of particles can be, and should be, regarded as the numerical error.

On the other hand, if we want to study long-term evolution of a star cluster, which takes place in the timescale much longer than the orbital timescale, the evolution of orbits of individual stars is driven by close encounters with other stars. In this case, the effect of small (or finite) number of particles is not an numerical error but what is there in real systems.

Thus, the required accuracy of pairwise force calculation depends on the nature of the problem. In the case of the study of the merging of two galaxies, the average error can be as large as 100% of the pairwise force, if the error is guaranteed to be random. The average error of interaction calculated by GRAPE-1 is less than 10%, which was good enough for many problems. In the number format used in GRAPE-1, the positions of particles are expressed in the 16-bit fixed-point format, so that the force between two nearby particles, both far from the origin of coordinates, is still expressed with sufficient accuracy. Also, the accumulation of the forces from different particles is done in the 48-bit fixed-point format, so that there is no loss of effective bits during the accumulation. Thus, the primary source of error with GRAPE-1 is the low-accuracy pairwise force calculation. It can be regarded as random error.

Strictly speaking, the error due to the short word format cannot always be regarded as random, since it introduces correlation both in space and time. One could eliminate this correlation by applying random coordinate transformation, but the quantitative study of such transformation has not done yet.

GRAPE-1 used the GPIB (IEEE-488) interface for the communication with the host computer. It was fast enough for the use with simple direct summation. However, when combined with the tree algorithm, faster

communication was necessary. GRAPE-1A used VME bus for communication, to improve the performance. The speed of GRAPE-1 and 1A was around 300 Mflops, which is around 1/10 of the speed of fastest vector supercomputers of the time. Hardware cost of them was around 1/1,000 of supercomputers, or roughly equal to the cost of low-end workstations. Thus, these first-generation GRAPEs offered price-performance two orders of magnitude better than that of general-purpose computers.

GRAPE-2 is similar to GRAPE-1A, but with much higher numerical accuracy. In order to achieve higher accuracy, commercial LSI chips for floating-point arithmetic operations such as TI SN74ACT8847 and Analog Devices ADSP3201/3202 were used. The pipeline of GRAPE-2 processes the three components of the interaction sequentially. So it accumulates one interaction in every three clock cycles. This approach was adopted to reduce the circuit size. Its speed was around 40 Mflops, but it is still much faster than workstations or minicomputers at that time.

GRAPE-3 was the first GRAPE computer with a custom LSI chip. The number format was the combination of the fixed point and logarithmic format similar to what were used in GRAPE-1. The chip was fabricated using 1 μm design rule by National Semiconductor. The number of transistors on chip was 110 K. The chip operated at 20 MHz clock speed, offering the speed of about 0.8 Gflops. Printed-circuit boards with eight chips were mass-produced, for the speed of 6.4 Gflops per board. Thus, GRAPE-3 was also the first GRAPE computer to integrate multiple pipelines into a system. Also, GRAPE-3 was the first GRAPE computer to be manufactured and sold by a commercial company. Nearly 100 copies of GRAPE-3 have been sold to more than 30 institutes (more than 20 outside Japan).

With GRAPE-4, a high-accuracy pipeline was integrated into one chip. This chip calculates the first time derivative of the force, so that fourth-order Hermite scheme [10] can be used. Here, again, the serialized pipeline similar to that of GRAPE-2 was used. The chip was fabricated using 1 μm design rule by LSI Logic. Total transistor count was about 400 K.

The completed GRAPE-4 system consisted of 1,728 pipeline chips (36 PCB boards each with 48 pipeline chips). It operated on 32 MHz clock, delivering the speed of 1.1 Tflops. Completed in 1995, GRAPE-4 was the first computer for scientific calculation to achieve the peak speed higher than 1 Tflops. Also, in 1995 and 1996, it was awarded the Gordon Bell Prize for peak performance, which is given to a real scientific calculation on a parallel computer with the highest performance. Technical details of machines from GRAPE-1 through GRAPE-4 can be found in [14] and references therein.

GRAPE-5 [9] was an improvement over GRAPE-3. It integrated two full pipelines which operate on 80 MHz clock. Thus, a single GRAPE-5 chip offered the speed eight times more than that of the GRAPE-3 chip, or the same speed as that of an eight-chip GRAPE-3 board. GRAPE-5 was awarded the 1999 Gordon Bell Prize for price-performance. The GRAPE-5 chip was fabricated with 0.35 μ m design rule by NEC.

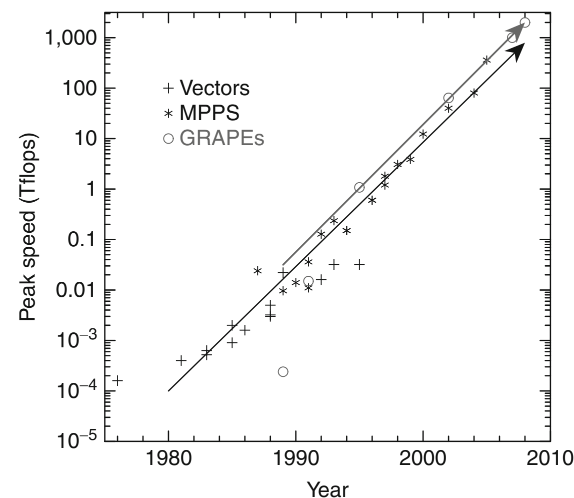
Table 1 summarizes the history of GRAPE project. Figure 2 shows the evolution of GRAPE systems and general-purpose parallel computers. One can see that evolution of GRAPE is faster than that of general-purpose computers.

The GRAPE-6 was essentially a scaled-up version of GRAPE-4 [15], with the peak speed of around 64 Tflops. The peak speed of a single pipeline chip was 31 Gflops. In comparison, GRAPE-4 consists of 1,728 pipeline chips, each with 600 Mflops. The increase of a factor of 50 in speed was achieved by integrating six pipelines into one chip (GRAPE-4 chip has one pipeline which needs three cycles to calculate the force from one particle) and using three times higher clock frequency. The advance of the device technology (from 1 μm to 0.25 μm) made these improvements possible. Figure 3 shows the processor chip delivered in early 1999. The six pipeline units are visible.

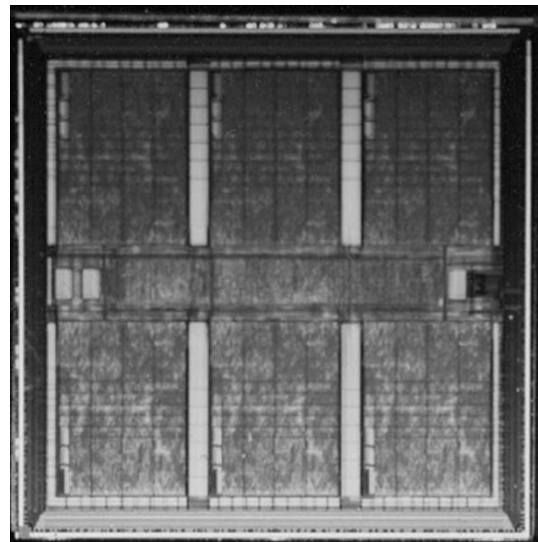
Starting with GRAPE-4, the concept of virtual multiple pipeline (VMP) is used. VMP is similar to simultaneous multithreading (SMT), in the sense that a single pipeline processor behaves as multiple processors. However, what is achieved is quite different. In the case of SMT, the primary gain is in the latency tolerance, since one can execute independent instructions with different threads. In the case of hardwired pipeline processors, there is no need to reduce the latency. With VMP, the bandwidth to the external memory is reduced, since the data of one particle which exerts force are shared by multiple virtual pipelines, each of which calculates the force on its own particle. This sharing of the

GRAPE. Table 1 History of GRAPE project

GRAPE-1	(89/4–89/10)	310 Mflops, low accuracy
GRAPE-2	(89/8–90/5)	50 Mflops, high accuracy(32 bit/64 bit)
GRAPE-1A	(90/4–90/10)	310 Mflops, low accuracy
GRAPE-3	(90/9–91/9)	18 Gflops, high accuracy
GRAPE-2A	(91/7–92/5)	230 Mflops, high accuracy
HARP-1	(92/7–93/3)	180 Mflops, high accuracy
		Hermite scheme
GRAPE-3A	(92/1–93/7)	8 Gflops/board
		some 80 copies are used all over the world
GRAPE-4	(92/7–95/7)	1 Tflops, high accuracy
		Some 10 copies of small machines
MD-GRAPE	(94/7–95/4)	1 Gflops/chip, high accuracy
		programmable interaction
GRAPE-5	(96/4–99/8)	5 Gflops/chip, low accuracy
GRAPE-6	(97/8–02/3)	64 Tflops, high accuracy



GRAPE. Fig. 2 The evolution of GRAPE and general-purpose parallel computers. The peak speed is plotted against the year of delivery. *Open circles, crosses*, and *stars* denote GRAPEs, vector processors, and parallel processors, respectively



GRAPE. Fig. 3 The GRAPE-6 processor chip

particle can be extended to physical multiple pipelines, as far as the total number of pipeline is not too large. Thus, special-purpose computers based on GRAPE-like pipeline have a unique advantage that their requirement of external memory bandwidth is much smaller than that of general-purpose computers with similar peak performance.

In the case of GRAPE-6, each of six physical pipelines is implemented as eight virtual pipelines. Thus, one GRAPE-6 chip calculates forces on 48 particles in parallel. The required memory bandwidth was 720 MB/s, for the peak speed of 31 Gflops. A traditional vector processor with the peak speed of 31 Gflops would require the memory bandwidth of 125

GB/s. Thus, GRAPE-6 requires the memory bandwidth around 1/200 of that of a traditional vector processor.

One processor board of GRAPE-6 housed 32 GRAPE-6 chips. Each GRAPE-6 chip has its own memory to store particles which exert the force. Thus, different processor chips on one processor board calculate the forces on the same 48 particles from different particles, and the partial results are summed up by an hardwired adder tree when the result is sent back to the host. Thus, the summation over 32 chips added only a small startup overhead (less than 1 μs) per one force calculation, which typically requires several milliseconds.

The completed GRAPE-6 system consisted of 64 processor boards, grouped into 4 clusters with 16 boards each. Within a cluster, 16 boards are organized in a 4×4 matrix, with 4 host computers. They are organized so that the effective communication speed is proportional to the number of host computers. In a simple configuration, the effective communication speed becomes independent of the number of host computers. The details of the network used in GRAPE-6 are given in [11].

Machines for Molecular Dynamics

Classical MD calculation is quite similar to astrophysical N-body simulations since, in both cases, we integrate the orbit of particles (atoms or stars) which interact with other particles with simple pairwise force. In the case of Coulomb force, the force law itself is the same as that of the gravitational force, and the calculation of Coulomb force can be accelerated by GRAPE hardware.

However, in MD calculations, the calculation cost of van der Waals force is not negligible, though van der Waals force decays much faster than the Coulomb force (r^{-7} compared to r^{-2}).

It is straightforward to design a pipelined processor which can handle particle–particle force given by some arbitrary function of the distance between particles. In GRAPE-2A and its successors, a combination of table lookup and polynomial approximation is used.

GRAPE-2A and MD-GRAPE were developed in the University of Tokyo, following these lines of idea. GRAPE-2A was built using commercial chips and MD-GRAPE used a custom-designed pipeline chip.

Another difference between astrophysical simulations and MD calculations is that in MD calculations, usually the periodic boundary condition is applied. Thus, we need some way to calculate Coulomb forces from image particles. The direct Ewald method is rather well suited for the implementation in hardware. In 1991, WINE-1 was developed. It is a pipeline to calculate the wave-space part of the direct Ewald method. The real-space part can be handled by GRAPE-2A or MD-GRAPE hardware.

In 1995, a group led by Toshikazu Ebisuzaki in RIKEN started to develop MDM [16], a massively parallel machine for large-scale MD simulations. Their primary goal was the simulation of protein molecules.

MDM consists of two special-purpose hardware, massively parallel version of MD-GRAPE (MDGRAPE-2) and that of WINE (WINE-2). The MDGRAPE-2 part consisted of 1,536 custom chips with four pipelines, for the theoretical peak speed of 25 Tflops. The WINE-2 part consists of 2,304 custom pipeline chips, for the peak speed of 46 Tflops.

The MDM effort was followed up by the development of MDGRAPE-3 [18], led by Makoto Taiji of RIKEN. MDGEAPE-3 achieved the peak speed of 1 Pflops in 2006.

Related Projects

The GRAPE project is not the first project to implement the calculation of pairwise force in particle-based simulations in hardware.

Delft Molecular Dynamics Processor [2] (DMDP) is one of the earliest efforts. It was completed in early 1980s. For the calculation of interaction between particles, it used the hardwired pipeline similar to that of GRAPE systems. However, in DMDP, time integration of orbits and other calculations are all done in the hardwired processors. Thus, in addition to the force calculation pipeline, DMDP had pipelines to update position, select particles for interaction calculation, and calculate diagnostics such as correlation function. FASTRUN [4] has the architecture similar to that of DMDP, but designed to handle more complex systems such as protein molecule.

To some extent, this difference in the designs of GRAPE computers and that of machines for molecular dynamics comes from the difference in the nature of the problem. In astronomy, the wide ranges in the number density of particles and timescale make it necessary to use adaptive schemes such as treecode and individual timesteps. With these schemes, the calculation cost per timestep per particle is generally higher than

that of fastest scheme optimized to shared timestep and near-uniform distribution of particles. The approach in which only the force calculation is done in hardware is more advantageous for astronomical N-body problems than for molecular dynamics.

Anton [17] is the latest effort to speed up the molecular dynamics simulation of proteins by specialized hardware. It is essentially the revival of the basic idea of DMDP, except that pipeline processors for operations other than force calculation were replaced by programmable parallel processors. It achieved the speed almost two orders of magnitude faster than that of general-purpose parallel computers for the simulation of protein molecules in water.

LSI Economics and GRAPE

GRAPE has achieved the cost performance much better than that of general-purpose computers. One reason for this success is simply that with GRAPE architecture, one can use practically all transistors for arithmetic units, without being limited by the memory wall problem. Another reason is the fact that arithmetic units can be optimized to their specific uses in the pipeline. For example, in the case of GRAPE-6, the subtraction of two positions is performed in 64-bit fixed point format, not in the floating-point format. Final accumulation is also done in fixed point. In addition, most of arithmetic operations to calculate the pairwise interactions are done in single precision. These optimizations made it possible to pack more than 200 arithmetic units into a single chip with less than 10 M transistors. The first microprocessor with fully pipelined double-precision floating-point unit, Intel 80860, required 1.2 M transistors for two (actually one and half) operations. Thus, the number of transistors per arithmetic unit of GRAPE is smaller by more than a factor of 10. When compared with more recent processors, the difference becomes even larger. The Fermi processor from NVIDIA integrates 512 arithmetic unit (adder and multiplier) with 3G transistors. Thus, it is five times less efficient than Intel 80860, and nearly 100 times less efficient than GRAPE-6. The difference in the power efficiency is even larger, because the requirement for the memory bandwidth is lower for GRAPE computers. As a result, performance per watt of GRAPE-6 chip, fabricated with the 250 nm design rule, is comparable to that of GPGPU chips fabricated with 40 nm design rule. Thus, as silicon technology advances, the relative advantage of special-purpose architecture such as GRAPE becomes bigger.

However, there is another economical factor. As the silicon semiconductor technology advances, the initial cost to design and fabricate custom chip increases. In 1990, the initial cost for a custom chip was around 100 K USD. By 2000, it has become higher than 1 M USD. By 2010, the initial cost of a 45 nm chip is around 10 M USD. Roughly speaking, initial cost has been increasing as $n^{0.7}$, where n is the number of transistors one can fit into a chip.

The total budget for GRAPE-4 and GRAPE-6 projects is 3 and 4 M USD, respectively. Thus, a similar budget had become insufficient by early 2000s. The whole point of special-purpose computer is to be able to outperform "expensive" supercomputers, with the price of 10–100 M USD. Even if a special-purpose computer is 100–1,000 times faster, it is not practical to spend the cost of a supercomputer for a special-purpose computer which can solve only a narrow range of problems.

There are several possible solutions. One is to reduce the initial cost by using FPGA (Field-Programmable Gate Array) chips. An FPGA chip consists of a number of "programmable" logic blocks (LBs) and also "programmable" interconnections. A LB is essentially a small lookup table with multiple inputs, augmented with one flip-flop and sometimes full-adder or more additional circuits. The lookup table can express any combinatorial logic for input data, and with flip-flop, it can be part of a sequential logic. Interconnection network is used to make larger and more complex logic, by connecting LBs. The design of recent FPGA chips has become much more complex, with large functional units like memory blocks and multiplier (typically 18 × 18 bit) blocks.

Because of the need for the programmability, the size of the circuit that can be fit into an FPGA chip is much smaller than that for a custom LSI, and the speed of the circuit is also slower. Roughly speaking, the price of an FPGA chip per logic gate is around 50 times higher than that of a custom chip with the same design rule. If the relative advantage of a specialized architecture is much larger than this factor of 50, its implementation based on FPGA chips can outperform general-purpose computers.

In reality, there are quite a number of projects to use FPGAs for scientific computing, but most of them

turned out to be not competitive with general-purpose computers. The primary reason for this result is the relative cost of FPGA discussed above. Since the logic gate of FPGAs is much more expensive than that of general-purpose computers, the design of a special-purpose computer with FPGA must be very efficient in gate usage. FPGA-based systems which use standard double- or single-precision arithmetic are generally not competitive with general-purpose computers. In order to be competitive, it is necessary to use much shorter word length. GRAPE architecture with reduced accuracy is thus an ideal target for FPGA-based approach. Several successful approaches have been reported [6, 8].

GRAPE-DR

Another solution for the problem of the high initial cost is to widen the application range by some way to justify the high cost. GRAPE-DR project [12] followed that approach.

With GRAPE-DR, the hardwired pipeline processor of previous GRAPE systems was replaced by a collection of simple SIMD programmable processors. The internal network and external memory interface was designed so that it could emulate GRAPE processor efficiently and could be used for several other important applications, including the multiplication of dense matrices.

GRAPE-DR is an acronym of "Greatly Reduced Array of Processor Elements with Data Reduction." The last part, "Data Reduction," means that it has an on-chip tree network which can do various reduction operations such as summation, max/min, and logical and/or.

The GRAPE-DR project was started in FY 2004, and finished in FY 2008. The GRAPE-DR processor chip consists of 512 simple processors, which can operate at the clock cycle of 500 MHz, for the 512 Gflops of single precision peak performance (256 Gflops double precision). It was fabricated with TSMC 90 nm process and the size is around 300 mm^2. The peak power consumption is around 60 W. The GRAPE-DR processor board houses four GRAPE-DR chips, each with its own local DRAM chips. It communicates with the host computer through Gen1 16-lane PCI-Express interface.

To some extent, the difference between GRAPE and GRAPE-DR is similar to that between traditional GPUs and GPGPUs. In both cases, hardwired pipelines are replaced by simple programmable processors. The main differences between GRAPE-DR and GPGPUs are (a) processor element of GRAPE-DR is much simpler, (b) external memory bandwidth of GRAPE-DR is much smaller, and (c) GRAPE-DR is designed to achieve near-peak performance in real scientific applications such as gravitational N-body simulation and molecular dynamics simulation, and also dense matrix multiplication. These differences made GRAPE-DR significantly more efficient in both transistor usage and power usage. GRAPE-DR chip, which was fabricated with 90 nm design rule and has 300 mm^2 area, integrates 512 processing elements. The NVIDIA Fermi chip, which is fabricated with 40 nm design rule and has $> 500\ mm^2$ area, integrates the same 512 processing elements. Thus, there is about a factor of 10 difference in the transistor efficiency. This difference resulted in more than a factor of 2 difference in the power efficiency.

Whether or not the approach like GRAPE-DR will be competitive with other approaches, in particular GPGPUs, is at the time of writing rather unclear. The reason is simply that the advantage of a factor of 10 is not quite enough, because of the difference in other factors, among which the most important is the development cycle. New GPUs are announced roughly every year, while it is somewhat unlikely that one develops the special-purpose computers every year, even if there is sufficient budget. In 5 years, general-purpose computers become ten times faster, and GPGPUs will also become faster by a similar factor. Thus, a factor of 10 advantage will disappear while the machine is being developed. On the other hand, the transistor efficiency of general-purpose computers, and that of GPUs, has been decreasing for the last 20 years and probably will continue to do so for the next 10 years or so. GRAPE-DR can retain its efficiency when it is implemented with more advanced semiconductor technology, since, as in the case of GRAPE, one can use the increased number of transistors to increase the number of processor element. Thus, it might remain competitive.

Future Directions

Future of Special-Purpose Processors

In hindsight, 1990s was a very good period for the development of special-purpose architecture such as GRAPE, because of two reasons. First, the semiconductor technology reached the point where many

floating-point arithmetic units can be integrated into a chip. Second, the initial design cost of a chip was still within the reach of fairly small research projects in basic science.

By now, semiconductor technology reached to the point that one could integrate thousands of arithmetic units into a chip. On the other hand, the initial design cost of a chip has become too high.

The use of FPGAs and the GRAPE-DR approach are two examples of the way to tackle the problem of increasing initial cost. However, unless one can keep increasing the budget, GRAPE-DR approach is not viable, simply because it still means exponential increase in the initial, and therefore total, cost of the project.

On the other hand, such increase in the budget might not be impossible, since the field of computational science as a whole is becoming more and more important. Even though a supercomputer is expensive, it is still much less expensive compared to, for example, particle accelerators or space telescopes. Of course, computer simulation cannot replace the real experiments of observations, but computer simulations have become essential in many fields of science and technology.

In addition, there are several technologies available in between FPGAs and custom chips. One is what is called "structured ASIC." It requires customization of typically just one metal layer, resulting in large reduction in the initial cost. The number of gates one can fit into the given silicon area falls between those of FPGAs and custom chips. Another possibility is just to use the technology one or two generations older.

Application Area of Special-Purpose Computers

Primary application area of GRAPE and GRAPE-DR has been the particle-based simulation, in particular that requires the evaluation of long-range interaction. It is suited to special-purpose computers because they are compute-intensive. In other words, the necessary bandwidth to the external memory is relatively small. Grid-based simulations based on schemes like finite-difference or finite-element methods are less compute-intensive and thus not suited to special-purpose computers.

However, the efficiency of large-scale parallel computers based on general-purpose microprocessors for these grid-based simulation has been decreasing rather quickly. There are two reasons for this decrease. One is the lack of the memory bandwidth. Currently, the memory bandwidth of microprocessors normalized by the calculation speed is around 0.3 bytes/flops, which is not enough for most of grid-based simulations. Even so, this ratio will become smaller and smaller in the future. The other reason is the latency of the communication between processors.

One possible solution for these two problems is to integrate the main memory, processor cores, and communication interface into a single chip. This integration gives practically unlimited bandwidth to the memory, and the communication latency is reduced by one or two orders of magnitude.

Obvious disadvantage of this approach is that the total amount of memory would be severely limited. However, in many application of grid-based calculation, very long time integrations of relatively small systems are necessary. Many of such applications requires memory much less than one TB, which can be achieved by using several thousand custom processors each with around 1 GB of embedded DRAM.

Bibliography

1. Aarseth SJ (1963) Dynamical evolution of clusters of galaxies, I. MN 126:223
2. Bakker AF, Gilmer GH, Grabow MH, Thompson K (1990) A special purpose computer for molecular dyanamics calculations. J Comput Phys 90:313
3. Barnes J, Hut P (1986) A hiearchical O(NlogN) force calculation algorithm. Nature 324:446
4. Fine R, Dimmler G, Levinthal C (1991) FASTRUN: a special purpose, hardwired computer for molecular simulation. Proteins Struct Funct Genet 11:242
5. Greengard L, Rokhlin V (1987) A fast algorithm for particle simulations. J Comput Phys 73:325
6. Hamada T, Fukushige T, Kawai A, Makino J (1999) PROGRAPE-1: a programmable, multi-purpose computer for many-body simulations. PASJ 52:943–995
7. Ito T, Makino J, Ebisuzaki T, Sugimoto D (1990) A special-purpose N-body machine GRAPE-1. Comput Phys Commun 60:187
8. Kawai A, Fukushige T (2006) $158/GFLOP Astrophysical N-body simulation with a reconfigurable add-in card and a hierarchical tree algorithm. In: Proceedings of SC06, ACM (Online)
9. Kawai A, Fukushige T, Makino J, Taiji M (2000) GRAPE-5: a special-purpose computer for N-body simulations. PASJ 52:659

10. Makino J, Aarseth SJ (1992) On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems. PASJ 44:141
11. Makino J, Fukushige T, Koga M, Namura K (2003) GRAPE-6: massively-parallel special-purpose computer for astrophysical particle simulations. PASJ 55:1163
12. Makino J, Hiraki K, Inaba M (2007) GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing. In: Proceedings of SC07, ACM (Online)
13. Makino J, Ito T, Ebisuzaki T (1990) Error analysis of the GRAPE-1 special-purpose N-body machine. PASJ 42:717
14. Makino J, Taiji M (1998) Scientific simulations with special-purpose computers – The GRAPE systems. Wiley, Chichester
15. Makino J, Taiji M, Ebisuzaki T, Sugimoto D (1997) GRAPE-4: a massively parallel special-purpose computer for collisional N-body simulations. ApJ 480:432
16. Narumi T, Susukita R, Ebisuzaki T, McNiven G, Elmegreen B (1999) Mol Simul 21:401
17. Shaw DE, Denero MM, Dror RO, Kuskin JS, Larson RH, Salmon JK, Young C, Batson B, Bowers KJ, Chao JC, Eastwood MP, Gagliardo J, Grossman JP, Ho CR, Ierardi DJ, Kolossváry I, Klepeis JL, Layman T, McLeavey C, Moraes MA, Mueller R, Priest EC, Shan Y, Spengler J, Theobald M, Towles B, Wang SC (2007) Anton: a special-purpose machine for molecular dynamics simulation. In: Proceedings of the 34th annual international symposium on computer architecture (ISCA '07), ACM, San Diego, pp 1–12
18. Taiji M, Narumi T, Ohno Y, Futatsugi N, Suenaga A, Takada N, Konagaya A (2003) Protein explorer: a petaflops special-purpose computer system for molecular dynamics simulations. In: The SC2003 Proceedings, IEEE, Los Alamitos, CD–ROM

Graph Algorithms

David A. Bader[1], Guojing Cong[2]
[1]Georgia Institute of Technology, Atlanta, GA, USA
[2]IBM, Yorktown Heights, NY, USA

Discussion

Parallel Graph Algorithms

Relationships in real-world situations can often be represented as graphs. Efficient parallel processing of graph problems has been a focus of many algorithm researchers. A rich collection of parallel graph algorithms have been developed for various problems on different models. The majority of them are based on the parallel random access machine (PRAM). PRAM is a shared-memory model where data stored in the global memory can be accessed by any processor. PRAM is synchronous, and in each unit of time, each processor either executes one instruction or stays idle.

Techniques

Graph problems are diverse. Given a graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, several techniques are frequently used in designing parallel graph algorithms. The basic techniques are described as follows (Detailed descriptions can be found in [24]).

Prefix Sum

Given a sequence of n elements $s_1, s_2, \cdots, s_n$ with a binary associative operator denoted by $\oplus$, the prefix sums of the sequence are the partial sums defined by

$$S_i = s_1 \oplus s_2 \oplus \cdots \oplus s_i, 1 \leq i \leq n$$

Using the balanced tree technique, the prefix sums of n elements can be computed in $O(\log n)$ time with $O(n/\log n)$ processors. Fast prefix sum is of fundamental importance to the design of parallel graph algorithms as it is frequently used in algorithms for more complex problems.

Pointer Jumping

Pointer jumping, also sometimes called path doubling, is useful for handling computation on rooted forests. For a rooted forest, there is a parent function P defined on the set of vertices. $P(r)$ is set to r when r does not have a parent. When finding the root of each tree, the pointer jumping technique updates the parent of each node by that node's grandparent, that is, set $P(r) = P(P(r))$. The algorithm runs in $O(\log n)$ time with $O(n)$ processors. Pointer jumping can also be used to compute the distance between each node to its root. Pointer jumping is used in several connectivity and tree algorithms (e.g., see [29, 31]).

Divide and Conquer

The divide-and-conquer strategy is recognized as a fundamental technique in algorithm design (not limited to parallel graph algorithms). The frequently used quicksort algorithm is based on divide and conquer. The strategy partitions the input into partitions of roughly equal size and recursively works on each partition concurrently. There is usually a final step to combine the solutions of the subproblems into a solution for the original problem.

Pipelining

Like divide-and-conquer, the use of pipelining is not limited to parallel graph algorithm design. It is of critical importance in computer hardware and software design. Pipelining breaks up a task into a sequence of smaller tasks (subtasks). Once a subtasks is complete and moves to the next stage, the ones following it can be processed in parallel. Insertion and deletion with 2–3 trees demonstrate the pipelining technique [27].

Deterministic Coin Tossing

Deterministic coin tossing was proposed by Cole and Vishkin [9] to break the symmetry of a directed cycle without using randomization. Consider the problem of finding a three-coloring of graph G. A k-coloring of G is a mapping $c : V \rightarrow \{0, 1, ..., k-1\}$ such that $c(i) \neq c(j)$ if $< i, j > \in E$. Initially, the cycle has a trivial coloring with $n = |V|$ colors. The apparent symmetry in the problem (i.e., the vertices cannot be easily distinguished) presents the major difficulty to reducing the number of colors needed for the coloring. Deterministic coin tossing uses the binary representation of an integer $i = \cdots i_k \cdots i_1 i_0$ to break the symmetry. For example, suppose t is the least significant bit position in which $c(i)$ and $c(S(i))$ differ ($S(i)$ is the successor of i in the cycle), then the new coloring for i can be set as $2t + c(i)_t$.

Accelerating Cascades

Accelerating cascades was presented together with deterministic coin tossing in [9] for the design of faster algorithms for list ranking and other problems. This technique combines two algorithms, one optimal and the other super fast, for the same problem to get a optimal and very fast algorithm. The general strategy is to start with the optimal algorithm to reduce the problem size and then apply the fast but nonoptimal algorithm.

Other Building Blocks for Parallel Graph Algorithms

List ranking determines the position, or rank, of the list items in a linked list. A generalized list-ranking problem is defined as follows. Let X be an array of n elements stored in arbitrary order. For each element i, let $X(i).value$ be its value and $X(i).next$ be the index of its successor. Then, for any binary associative operator $\oplus$, compute $X(i).prefix$ such that $X(head).prefix = X(head).value$ and $X(i).prefix = X(i).value \oplus X(predecessor).prefix$, where *head* is the first element of the list, i is not equal to *head*, and *predecessor* is the node preceding i in the list. Pointer jumping can be applied to solve the list-ranking problem. An optimal list-ranking algorithm is given in [10].

The Euler tour technique is another of the basic building blocks for designing parallel algorithms, especially for tree computations. For example, postorder/preorder numbering, computing the vertex level, computing the number of descendants, etc., can be done work-time optimally on EREW PRAM by applying the Euler tour technique. As suggested by its name, the power of the Euler tour technique comes from defining a Eulerian circuit on the tree. In Tarjan and Vishkin's biconnected components paper [31] that originally introduced the Euler tour technique, the input to their algorithm is an edge list with the cross-pointers between twin edges $< u, v >$ and $< v, u >$ established. With these cross-pointers it is easy to derive an Eulerian circuit. The Eulerian circuit can be treated as a linked list, and by assigning different values to each edge in the list, list ranking can be used for many tree computation. For example, when rooting a tree, the value 1 is associated with each edge. After list ranking, simply inspecting the list rank for $< u, v >$ and $< v, u >$ can set the correct parent relationship for u and v.

Tree contraction systematically shrinks a tree into a single vertex by successively shrinking parts of the tree. It can be used to solve the expression evaluation problem. It is also used in other algorithm, for example, computing the biconnected components [31].

Classical Algorithms

The use of the basic techniques are demonstrated in several classical graph algorithms for spanning tree, minimum spanning tree, and biconnected components.

Various deterministic and randomized techniques have been given for solving the spanning tree problem (and the closely related connected components problem) on PRAM models. A brief survey of these algorithms can be found in [3]. The Shiloach–Vishkin algorithm (SV) algorithm is representative of several connectivity algorithms in that it adapts the widely used graft-and-shortcut approach. Through carefully designed grafting schemes, the algorithm achieves complexities of $O(\log n)$ time and $O((m + n) \log n)$

work under the arbitrary CRCW PRAM model. The algorithm takes an edge list as input and starts with n isolated vertices and m processors. Each processor P_i ($1 \leq i \leq m$) inspects edge $e_i = < v_{i_1}, v_{i_2} >$ and tries to graft vertex v_{i_1} to v_{i_2} under the constraint that $v_{i_1} < v_{i_2}$. Grafting creates $k \geq 1$ connected components in the graph, and each of the k components is then shortcut to to a single super-vertex. Grafting and shortcutting are iteratively applied to the reduced graphs $G' = (V', E')$ (where V' is the set of super-vertices and E' is the set of edges among super-vertices) until only one super-vertex is left.

Minimum spanning tree (MST) is one of the most studied combinatorial problems with practical applications. While several theoretic results are known for solving MST in parallel, many are considered impractical because they are too complicated and have large constant factors hidden in the asymptotic complexity. See for a survey of MST algorithms. Many parallel algorithms are based on the Borůvka algorithm. Borůvka's algorithm is comprised of Borůvka iterations that are used in many parallel MST algorithms. A Borůvka iteration is characterized by three steps: *find-min*, *connected-components*, and *compact-graph*. In *find-min*, for each vertex v the incident edge with the smallest weight is labeled to be in the MST; *connect-components* identifies connected components of the induced graph with the labeled MST edges; *compact-graph* compacts each connected component into a single supervertex, removes self-loops and multiple edges, and relabels the vertices for consistency.

A connected graph is said to be *separable* if there exists a vertex v such that removal of v results in two or more connected components of the graph. Given a connected, undirected graph G, the biconnected components problem finds the maximal-induced subgraphs of G that are not *separable*. Tarjan [30] presents an optimal $O(n+m)$ algorithm that finds the biconnected components of a graph based on depth-first search (DFS). Eckstein [17] gave the first parallel algorithm that takes $O(d \log^2 n)$ time with $O((n+m)/d)$ processors on CREW PRAM, where d is the diameter of the graph. Tarjan and Vishkin [31] present an $O(\log n)$ time algorithm on CRCW PRAM that uses $O(n+m)$ processors. This algorithm utilizes many of the fundamental primitives including prefix sum, list ranking, sorting, connectivity, spanning tree, and tree computations.

Communication Efficient Graph Algorithms

Communication-efficient parallel algorithms were proposed to address the "bottleneck of processor-to-processor communication" (e.g., see [15]). Goodrich [19] presented a communication-efficient sorting algorithm on weak-CREW BSP that runs in $O(\log n / \log(h+1))$ communication rounds (with at most h data transported by each processor in each round) and $O((n \log n)/p)$ local computation time, for $h = \Theta(n/p)$. Goodrich's sorting algorithm is frequently used in communication-efficient graph algorithms. Dehne et al. designed an efficient list-ranking algorithm for coarse-grained multicomputers (CGM) and BSP that takes $O(\log p)$ communication rounds with $O(n/p)$ local computation. In the same study, a series of communication-efficient graph algorithms such as connected components, ear decomposition, and biconnected components are presented using the list-ranking algorithm as a building block. On the BSP model, Adler et al. [1] presented a communication-optimal MST algorithm. The list-ranking algorithm and the MST algorithm take similar approaches to reduce the number of communication rounds. They both start by simulating several (e.g., $O(\log p)$ or $O(\log \log p)$) steps of the PRAM algorithms on the target model to reduce the input size so that it fits in the memory of a single node. A sequential algorithm is then invoked to process the reduced input of size $O(n/p)$, and finally the result is broadcast to all processors for computing the final solution.

Practical Implementation

Many real-world graphs are large and sparse. These instances are especially hard to process due to the characteristics of the workload. Although fast theoretic algorithms exist in the literature, large and sparse graph problems are still challenging to solve in practice. There remains a significant gap between algorithmic model and architecture. The mismatch between memory access pattern and cache organization is the most outstanding barrier to high-performance graph analysis on current systems.

Graph Workload

Compared with traditional scientific applications, graph analysis is more memory intensive. Graph algorithms put tremendous pressure on the memory subsystem

to deliver data to the processor. Table 1 shows the percentages of memory instructions executed in the SPLASH2 benchmarks [34] and several graph algorithms. SPLASH2 represents typical scientific applications for shared-memory environments. On IBM Power5, on average, about 10% more memory instructions are executed in the graph algorithms. For biconnected components, the number is over 59%.

For memory-intensive applications, the locality behavior is especially crucial to the performance. Unfortunately, most graph algorithms exhibit erratic memory access patterns that result in poor performance, as shown in Table 2. Table 2 is the cycles per instruction (CPI) construction [13] for three graph algorithms on IBM Power5. The algorithms studied are betweenness centrality (BC), biconnected components (BiCC), and minimum spanning tree (MST). CPI construction attributes in percentage cycles to categories such as completion, instruction cache miss penalty, and stall. In Table 2, for all algorithms, significant amount of cycles are spent on pipeline stalls. About 60–70% of the cycles are wasted on the load-store unit stalls, and more than 50% of the cycles are spent on stalls due to data cache misses. Table 2 clearly shows that graph algorithms perform poorly on current cache-based architectures, and the culprit is the memory access pattern. The floating-point stalls (FPU STL) column is revealing about one other prominent feature of graph algorithms. FPU STL is the percentage of cycles wasted on floating-point unit stalls. In Table 2, BiCC and MST do not incur any FPU stalls. Unfortunately, this does not mean that the workload fully utilizes the FPU. Instead, as there are no floating-point operations in these algorithms, the elaborately designed floating point units lay idle. CPI construction shows that graph workloads spent most of the time waiting for data to be delivered, and there are not many other operations to hide the long latency to main memory. In fact, of the three algorithms, only BC incurs execution of a few floating-point instructions.

Implementation on Shared-Memory Machines

It is relatively straightforward to map PRAM algorithms on to shared-memory machines such as symmetric multiprocessors (SMPs), multi-core and many core systems. While these systems are of shared-memory architecture, they are by no means the PRAM used in theoretical work – synchronization cannot be taken for granted, memory bandwidth is limited, and performance requires a high degree of locality. Practical design choices need to be made to achieve high performance on such systems.

Adapting to the Available Parallelism

Nick's Class ($\mathcal{NC}$) is defined as the set of all problems that run in polylog-time with a polynomial number of processors. Whether a problem P is in $\mathcal{NC}$ is a fundamental question. The PRAM model assumes an

Graph Algorithms. Table 1 Percentages of load-store instructions for the SPLASH2 benchmark and the graph problems. ST, MST, BiCC, and CC stand for spanning tree, minimum spanning tree, biconnected components, and connected components, respectively

Benchmark	SPLASH2				Graph problems			
	Barnes	Cholesky	Ocean	Raytrace	ST	MST	BiCC	CC
Load	20.3%	20.6%	21.4%	25.1%	45.6%	28.6%	40.4%	44.5%
Store	15.6%	5.2%	17.8%	9.5%	2%	15.2%	19.2%	1.4%
Load+store	35.9%	25.8%	39.2%	34.6%	47.6%	43.8%	59.6%	45.9%

Graph Algorithms. Table 2 CPI construction for three graph algorithms. Base cycles are for "useful" work. The "Stall" columns show the percentages of cycles on pipeline stalls followed by stalls due to load-store unit, data cache miss, floating-point unit, fix point unit, respectively

Algorithm	Base	GCT	Stall	LSU STL	DCache STL	FPU STL	FXU STL
BC	0.099	0.011	0.888	0.797	0.558	0.030	0.016
BiCC	0.170	0.066	0.762	0.600	0.445	0.000	0.060
MST	0.106	0.037	0.855	0.713	0.600	0.000	0.015

unlimited number of processors and explores the maximum inherent parallelism of P. Acknowledging the practical restriction of limited parallelism provided by real computers, Kruskal et al. [26] argued that non-polylogarithmic time algorithms (e.g., sublinear time algorithms) could be more suitable than polylog algorithms for implementation with practically large input size. $\mathcal{EP}$ (short for *efficient parallel*) algorithms, by the definition in [26], is the class of algorithms that achieve a polynomial reduction in running time with a polylogarithmic inefficiency. With $\mathcal{EP}$ algorithms, the design focus is shifted from reducing the complexity factors to solving problems of realistic sizes efficiently with a limited number of processors.

Reducing Synchronization

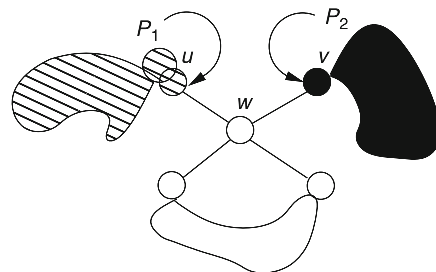
When adapting a PRAM algorithm to shared-memory machines, a thorough understanding of the algorithm usually suffices to eliminate unnecessary synchronization such as barriers. Reducing synchronization thus is an implementation issue. Asynchronous algorithm design, however, is more aggressive in reducing synchronization. It sometimes allows nondeterministic intermediate results but deterministic solutions.

In a parallel environment, to ensure correct final results oftentimes a total ordering on all the events is not necessary, and a partial ordering in general suffices. Relaxed constraints on ordering reduce the number of synchronization primitives in the algorithm.

Bader and Cong presented a largely asynchronous spanning tree algorithm in [3] that employs a constant number of barriers. The spanning tree algorithm for shared-memory machines has two main steps: (1) stub spanning tree and (2) work-stealing graph traversal. In the first step, one processor generates a stub spanning tree, that is, a small portion of the spanning tree by randomly walking the graph for $O(p)$ steps. The vertices of the stub spanning tree are evenly distributed into each processor's queue, and each processor in the next step will traverse from the first element in its queue. After the traversals in step 2, the spanning subtrees are connected to each other by this stub spanning tree. In the graph traversal step, each processor traverses the graph (by coloring the nodes) similar to the sequential algorithm in such a way that each processor finds a subgraph of the final spanning tree. Work-stealing is used to balance the load for graph traversal.

One problem related to synchronization is that there could be portions of the graph traversed by multiple processors and be in different subgraphs of the spanning tree. The immediate remedy is to synchronize using either locks or barriers. With locks, coloring the vertex becomes a critical section, and a processor can only enter the critical section when it gets the lock. Although the nondeterministic behavior is now prevented, it does not perform well on large graphs due to an excessive number of locking and unlocking operations.

In the proposed algorithm, no barriers are introduced in graph traversal. The algorithm runs correctly without, barriers even when two or more processors color the same vertex. In this situation, each processor will color the vertex and set as its parent the vertex it has just colored. Only one processor succeeds at setting the vertex's parent to a final value. For example, using Fig. 1, processor P_1 colored vertex u, and processor P_2 colored vertex v, and at a certain time they both find w unvisited and are now in a race to color vertex w. It makes no difference which processor colored w last because w's parent will be set to either u or v (and it is legal to set w's parent to either of them; this will not change the validity of the spanning tree, only its shape). Further, this event does not create cycles in the spanning tree under sequential consistency model. Both P_1 and P_2 record that w is connected to each processor's own tree. When each of w's unvisited children are visited by various processors, its parent will be set to w, independent of w's parent.



Graph Algorithms. Fig. 1 Two processors P_1 and P_2 see vertex w as unvisited, so each is in a race to color w and set w's parent pointer. The shaded area represents vertices colored by P_1, the black area represents those marked by P_2, and the white area contains unvisited vertices

Choosing Between Barriers and Locks

Locks and barriers are two major types of synchronization primitives. In practice, the choice of using locks or barriers may not be very clear. Take the "graft and shortcut" spanning tree algorithm for example. For graph $G = (V, E)$ represented as an edge list, the algorithm starts with n isolated vertices and $2m$ processors. For edge $e_i = \langle u, v \rangle$, processor P_i $(1 \le i \le m)$ inspects u and v, and if $v < u$, it grafts vertex u to v and labels e_i to be a spanning tree edge. The problem here is that for a certain vertex v, its multiple incident edges could cause grafting v to different neighbors, and the resulting tree may not be valid. To ensure that v is only grafted to one of the neighbors, locks can be used. Associated with each vertex v is a flag variable protected by a lock that shows whether v has been grafted. In order to graft v a processor has to obtain the lock and check the flag, thus race conditions are prevented. A different solution uses barriers [31] in a two-phase election. No checking is needed when a processor grafts a vertex, but after all processors are done (ensured with barriers), a check is performed to determine which one succeeds and the corresponding edge is labeled as a tree edge. Whether to use a barrier or lock is dependent on the algorithm design as well as the barrier and lock implementations. Locking typically introduces large memory overhead. When contention among processors is intense, the performance degrades significantly.

Cache Friendly Design

The increasing speed difference between processor and main memory makes cache and memory access patterns important factors for performance. The fact that modern processors have multiple levels of memory hierarchy is generally not reflected by most of the parallel models. As a result, few parallel algorithm studies have touched on the cache performance issue. The SMP model proposed by Helman and JáJá is the first effort to model the impact of memory access and cache over an algorithm's performance [21]. The model forces an algorithm designer to reduce the number of noncontiguous memory accesses. However, it does not give hints to the design of cache-friendly parallel algorithms.

Chiang et al. [7] presented a *PRAM simulation* technique for designing and analyzing efficient external-memory (sequential) algorithms for graph problems. This technique simulates the PRAM memory by keeping a task array of $O(N)$ on disk. For each PRAM step, the simulation sorts a copy of the contents of the PRAM memory based on the indices of the processors for which they will be operands, and then scans this copy and performs the computation for each processor being simulated. The following can be easily shown:

Theorem 1 *Let A be a PRAM algorithm that uses N processors and $O(N)$ space and runs in time T. Then A can be simulated in $O(T \cdot sort(N))$ I/Os* [7].

Here, $sort(N)$ represents the optimal number of I/Os needed to sort N items striped across the disks, and $scan(N)$ represents the number of I/Os needed to read N items striped across the disks. Specifically,

$$sort(x) = \frac{x}{DB} \log_{\frac{M}{B}} \frac{x}{B}$$

$$scan(x) = \frac{x}{DB}$$

where M = # of items that can fit into main memory, B = # of items per disk block, and D = # of disks in the system.

A similar technique can be applied to the cache-friendly parallel implementation of PRAM algorithms for large inputs. I/O efficient algorithms exhibit good spatial locality behavior that is critical to good cache performance. Instead of having one processor simulate the PRAM step, $p \ll n$ processors may perform the simulation concurrently. The simulated PRAM implementation is expected to incur few cache block transfers between different levels. For small input sizes, it would not be worthwhile to apply this technique as most of the data structures can fit into cache. As the input size increases, the cost to access memory becomes more significant, and applying the technique becomes beneficial.

Algorithmic Optimizations

For most problems, parallel algorithms are inherently more complicated than the sequential counterparts, incurring large overheads with many algorithm steps. Instead of lowering the asymptotic complexities, in many cases, reducing the constant factors improves performance. Cong and Bader demonstrates the benefit of such optimizations with their biconnected components algorithm [11].

The algorithm eliminates edges that are not essential in computing the biconnected components. For any

input graph, edges are first eliminated before the computation of biconnected components is done so that at most $min(m, 2n)$ edges are considered. Although applying the filtering algorithm does not improve the asymptotic complexity, in practice, the performance of the biconnected components algorithm can be significantly improved.

An edge e is considered as *nonessential* for biconnectivity if removing e does not change the biconnectivity of the component to which it belongs. Filtering out *nonessential* edges when computing biconnected components (these edges are put back in later) yields performance advantages. The Tarjan–Vishkin algorithm (TV) is all about finding the equivalence relation $R_c'^*$ [24, 31]. Of the three conditions for R_c', it is trivial to check for condition 1 which is for a tree edge and a non-tree edge. Conditions 2 and 3, however, are for two tree edges, and checking involves the computation of *high* and *low* values. To compute *high* and *low*, every non-tree edge of the graph is inspected, which is very time consuming when the graph is not extremely sparse. The fewer edges the graph has, the faster the *Low-high* step. Also, when building the auxiliary graph, the fewer edges in the original graph means the smaller the auxiliary graph and the faster the *Label-edge* and *Connected-components* steps.

Combining the filtering algorithm for eliminating *nonessential* edges and TV, the new biconnected components algorithm runs in $max(O(d), O(\log n))$ time with $O(n)$ processors on CRCW PRAM, where d is the diameter of the graph. Asymptotically, the new algorithm is not faster than TV. In practice, however, parallel speedups upto 4 with 12 processors are achieved on SUN Enterprise 4500 using the filtering technique.

Implementation on Multithreaded Architectures

Graph algorithms have been observed to run well on multi-threaded architectures such as the CRAY MTA-2 [5] and its successor, the Cray XMT. The Cray MTA [14] is a flat, shared-memory multiprocessor system. All memory is accessible and equidistant from all processors. There is no local memory and no data caches. Parallelism, and not caches, is used to tolerate memory and synchronization latencies.

An MTA processor consists of 128 hardware streams and one instruction pipeline. Each stream can have up to 8 outstanding memory operations. Threads from the same or different programs are mapped to the streams by the runtime system. A processor switches among its streams every cycle, executing instructions from non-blocked streams in a fair manner. As long as one stream has a ready instruction, the processor remains fully utilized.

Bader et al. compared the performance of list-ranking algorithm on SMPs and MTA [5]. For list ranking, they used two classes of list to test the algorithms: *Ordered* and *Random*. Ordered places each element in the array according to its rank; thus, node i is the ith position of the array and its successor is the node at position $(i + 1)$. Random places successive elements randomly in the array. Since the MTA maps contiguous logical addresses to random physical addresses, the layout in physical memory for both classes is similar, the performance on the MTA is independent of order. This is in sharp contrast to SMP machines which rank Ordered lists much faster than Random lists. On the SMP, there is a factor of 3–4 difference in performance between the best case (an ordered list) and the worst case (a randomly-ordered list). On the ordered lists, the MTA is an order of magnitude faster than the SMP, while on the random list, the MTA is approximately 35 times faster.

Implementation on Distributed Memory Machines

As data partitioning and explicit communication are required, implementing highly irregular algorithms is hard on distributed-memory machines. As a result, although many fast theoretic algorithms exist in the literature, few experimental results are known. As for performance, the adverse impact of irregular accesses is magnified in the distributed-memory environment when memory requests served by remote nodes experience long network latency. Two studies have demonstrated reasonable parallel performance with distributed-memory machines [28, 35]. Both studies implement parallel breadth-first search (BFS), one on BlueGene/L [2] and the other on CELL/BE [22]. The CELL architecture resembles a distributed-memory setting as explicit data transfer is necessary between the local storage on an SPE and the main memory. Neither study establishes a strong evidence for fast execution of parallel graph algorithms on distributed-memory systems. The individual CPU in BlueGene and the PPE

of CELL are weak compared with other Power processors or the SPE. It is hard to establish a meaningful baseline to compare the parallel performance against. Indeed, in both studies, either only wall clock times or speedups compared with other reference architectures are reported.

Partitioned global address space (PGAS) languages such as UPC and X10 [6, 32] have been proposed recently that present a shared-memory abstraction to the programmer for distributed-memory machines. They allow the programmer to control the data layout and work assignment for the processors. Mapping shared-memory graph algorithms onto distributed-memory machines is straightforward with PGAS languages.

Performance wise, straightforward PGAS implementation for irregular graph algorithms does not usually achieve high performance due to the aggregate startup cost of many small messages. Cong, Almasi, and Saraswat presented their study in optimizing the UPC implementation of graph algorithm in [12]. They improve both the communication efficiency and the cache performance of the algorithm through improving the locality behavior.

Some Experimental Results

Greiner [20] implemented several connected components algorithms using NESL on the Cray Y-MP/C90 and TMC CM-2. Hsu, Ramachandran, and Dean [23] also implemented several parallel algorithms for connected components. They report that their parallel code runs 30 times slower on a MasPar MP-1 than Greiner's results on the Cray, but Hsu et al.'s implementation uses one-fourth of the total memory used by Greiner's approach. Krishnamurthy et al. [25] implemented a connected components algorithm (based on Shiloach-Vishkin [29]) for distributed memory machines. Their code achieved a speedup of 20 using a 32-processor TMC CM-5 on graphs with underlying 2D and 3D regular mesh topologies, but virtually no speedup on sparse random graphs. Goddard, Kumar, and Prins [18] implemented a connected components algorithm for a mesh-connected SIMD parallel computer, the 8192-processor MasPar MP-1. They achieve a maximum parallel speedup of less than two on a random graph with 4,096 vertices and about one-million edges. For a random graph with 4,096 vertices and fewer than a half-million edges, the parallel implementation was slower than the sequential code.

Chung and Condon [8] implemented a parallel minimum spanning tree (MST) algorithm based on Borůvka's algorithm. On a 16-processor CM-5, for geometric graphs with 32,000 vertices and average degree 9 and graphs with fewer vertices but higher average degree, their code achieved a parallel speedup of about 4, on 16-processors, over the sequential Borůvka's algorithm, which was 2–3 times slower than their sequential Kruskal algorithm.

Dehne and Götz [16] studied practical parallel algorithms for MST using the BSP model. They implemented a dense Borůvka parallel algorithm, on a 16-processor Parsytec CC-48, that works well for sufficiently dense input graphs. Using a fixed-sized input graph with 1,000 vertices and 400,000 edges, their code achieved a maximum speedup of 6.1 using 16 processors for a random dense graph. Their algorithm is not suitable for sparse graphs.

Woo and Sahni [33] presented an experimental study of computing biconnected components on a hypercube. Their test cases are graphs that retain 70 and 90% edges of the complete graphs, and they achieved parallel efficiencies up to 0.7 for these dense inputs. The implementation uses adjacency matrix as input representation, and the size of the input graphs is limited to less than 2K vertices.

Bader and Cong presented their studies [3, 4, 11] of the spanning tree, minimum spanning tree, and biconnected components algorithms on SMPs. They achieved reasonable parallel speedups on the large, sparse inputs compared with the best sequential implementations.

Bibliography

1. Adler M, Dittrich W, Juurlink B, Kutyłowski M, Rieping I (1998) Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In: SPAA'98: proceedings of the tenth annual ACM symposium on parallel algorithms and architectures. ACM, New York, pp 27–36
2. Allen F, Almasi G et al (2001) Blue Gene: a vision for protein science using a petaflop supercomputer. IBM Syst J 40(21):310–327
3. Bader DA, Cong G (2004) A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: Proceedings of the 18th international parallel and distributed processing symposium (IPDPS 2004), Santa Fe

4. Bader DA, Cong G (2004) Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In: Proceedings of the 18th international parallel and distributed processing symposium (IPDPS 2004), Santa Fe
5. Bader DA, Cong G, Feo J (2005) On the architectural requirements for efficient execution of graph algorithms. In: Proceeding of the 2005 international conference on parallel processing, Oslo, pp 547–556
6. Charles P, Donawa C, Ebcioglu K, Grothoff C, Kielstra A, Praun CV, Saraswat V, Sarkar V (2005) X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 2005 ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA), San Diego, pp 519–538
7. Chiang Y-J, Goodrich MT, Grove EF, Tamassia R, Vengroff DE, Vitter JS (1995) External-memory graph algorithms. In: Proceedings of the 1995 symposium on discrete algorithms, San Francisco, pp 139–149
8. Chung S, Condon A (1996) Parallel implementation of Borůvka's minimum spanning tree algorithm. In: Proceedings of the tenth international parallel processing symposium (IPPS'96), Honolulu, pp 302–315
9. Cole R, Vishkin U (1986) Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In: STOC'86: proceedings of the eighteenth annual ACM symposium on theory of computing. ACM, New York, pp 206–219
10. Cole R, Vishkin U (1991) Approximate parallel scheduling. part II: applications to logarithmic-time optimal graph algorithms. Info Comput 92:1–47
11. Cong G, Bader DA (2005) An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In: Proceedings of the 19th international parallel and distributed processing symposium (IPDPS 2005), Denver
12. Cong G, Almasi G, Saraswat V (2010) Fast PGAS implementation of distributed graph algorithms. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis (SC2010), SC'10. IEEE Computer Society, Washington, DC, pp 1–11
13. CPI analysis on Power5. On line, 2006. http://www.ibm.com/developerworks/linux/library/pacpipower1/index.html
14. Cray, Inc. (2005) The CRAY MTA-2 system. www.cray.com/products/programs/mta_2/
15. Culler DE, Dusseau AC, Martin RP, Schauser KE (1993) Fast parallel sorting under LogP: from theory to practice. In: Portability and performance for parallel processing. Wiley, New York, pp 71–98 (Chap 4)
16. Dehne F, Götz S (1998) Practical parallel algorithms for minimum spanning trees. In: Workshop on advances in parallel and distributed systems, West Lafayette, pp 366–371
17. Eckstein DM (1979) BFS and biconnectivity. Technical Report 79–11, Department of Computer Science, Iowa State University of Science and Technology, Ames
18. Goddard S, Kumar S, Prins JF (1997) Connected components algorithms for mesh-connected parallel computers. In: Bhatt SN (ed) Parallel algorithms: third DIMACS implementation challenge, 17–19 October 1994. DIMACS series in discrete mathematics and theoretical computer science, vol 30. American Mathematical Society, Providence, pp 43–58
19. Goodrich MT (1996) Communication-efficient parallel sorting. In STOC'96: proceedings of the twenty-eighth annual ACM symposium on theory of computing. ACM, New York, pp 247–256
20. Greiner J (1994) A comparison of data-parallel algorithms for connected components. In: Proceedings of the sixth annual symposium on parallel algorithms and architectures (SPAA-94), Cape, May, pp 16–25
21. Helman DR, JáJá J (1999) Designing practical efficient algorithms for symmetric multiprocessors. In: Algorithm engineering and experimentation (ALENEX'99). Lecture notes in computer science, vol 1619. Springer, Baltimore, pp 37–56
22. Hofstee HP (2005) Power efficient processor architecture and the cell processor. In: International symposium on high-performance computer architecture, San Francisco, pp 258–262
23. Hsu T-S, Ramachandran V, Dean N (1997) Parallel implementation of algorithms for finding connected components in graphs. In: Bhatt SN (ed) Parallel algorithms: third DIMACS implementation challenge, 17–19 October 1994. DIMACS series in discrete mathematics and theoretical computer science, vol 30. American Mathematical Society, Providence, pp 23–41
24. JáJá J (1992) An introduction to parallel algorithms. Addison-Wesley, New York
25. Krishnamurthy A, Lumetta SS, Culler DE, Yelick K (1997) Connected components on distributed memory machines. In Bhatt SN (ed) Parallel algorithms: third DIMACS implementation challenge, 17–19 October 1994. DIMACS series in discrete mathematics and theoretical computer science, vol 30. American Mathematical Society, Providence, pp 1–21
26. Kruskal CP, Rudolph L, Snir M (1990) Efficient parallel algorithms for graph problems. Algorithmica 5(1):43–64
27. Paul WJ, Vishkin U, Wagener H (1983) Parallel dictionaries in 2–3 trees. In: Tenth colloquium on automata, languages and programming (ICALP), Barcelona. Lecture notes in computer science. Springer, Berlin, pp 597–609
28. Scarpazza DP, Villa O, Petrini F (2008) Efficient breadth-first search on the Cell/BE processor. IEEE Trans Parallel Distr Syst 19(10):1381–1395
29. Shiloach Y, Vishkin U (1982) An O(logn) parallel connectivity algorithm. J Algorithms 3(1):57–67
30. Tarjan RE (1972) Depth-first search and linear graph algorithms. SIAM J Comput 1(2):146–160
31. Tarjan RE, Vishkin U (1985) An efficient parallel biconnectivity algorithm. SIAM J Comput 14(4):862–874
32. Unified Parallel C, URL: http://en.wikipedia.org/wiki/Unified_Parallel_C
33. Woo J, Sahni S (1991) Load balancing on a hypercube. In: Proceedings of the fifth international parallel processing symposium, Anaheim. IEEE Computer Society, Los Alamitos, pp 525–530
34. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The SPLASH-2 programs: characterization and methodological

considerations. In: Proceedings of the 22nd annual international symposium computer architecture, pp 24–36
35. Yoo A, Chow E, Henderson K, McLendon W, Hendrickson B, Çatalyürek ÜV (2005) A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In: Proceedings of supercomputing (SC 2005), Seattle

Graph Analysis Software

►SNAP (Small-World Network Analysis and Partitioning) Framework

Graph Partitioning

Bruce Hendrickson
Sandia National Laboratories, Albuquerque, NM, USA

Definition

Graph partitioning is a technique for dividing work amongst processors to make effective use of a parallel computer.

Discussion

When considering the data dependencies in a parallel application, it is very convenient to use concepts from graph theory. A *graph* consists of a set of entities called *vertices*, and a set of pairs of entities called *edges*. The entities of interest in parallel computing are small units of computation that will be performed on a single processor. They might be the work performed to update the state of a single atom in a molecular dynamics simulation, or the work required to compute the contribution of a single row of a matrix to a matrix-vector multiplication. Each such work unit will be a vertex in the graph which describes the computation. If two units have a data dependence between them (i.e., the output of one computation is required as input to the other), then there will be an edge in the graph that joins the two corresponding vertices.

For a computation to perform efficiently on a parallel machine each of the P processors needs to have about the same amount of work to perform, and the amount of inter-processor communication must be small. These two conditions can be viewed in terms of the computational graph. The vertices of the graph (signifying units of work) need to be divided into P sets with about the same number of vertices in each. Additionally, the number of edges that connect vertices in two different sets needs to be kept small since these will reflect the need for interprocessor communication. This problem is known as *graph partitioning* and is an important approach to the parallelization of many applications.

More generally, the vertices of the graph can have weights associated with them, reflecting different amounts of computation, and the edges can also have weights corresponding to different quantities of communication. The graph partitioning problem involves dividing the set of vertices into P sets with about the same amount of total vertex weight, while keeping small the total weight of edges that cross between partitions. This problem is known to be NP-hard, but a number of heuristics have been devised that have proven to be effective for many parallel computing applications. Several software tools have been developed for this problem, and they are an important piece of the parallel computing ecosystem. Important algorithms and tools are discussed below.

Parallel Computing Applications of Graph Partitioning

Graph partitioning is a useful technique for parallelizing many scientific applications. It is appropriate when the calculation consists of a series of steps in which the computational structure and data dependencies do not vary much. Under such circumstances the expense of partitioning is rewarded by improved parallel performance for many computational steps.

The partitioning model is most applicable for bulk synchronous parallel applications in which each step consists of local computation followed by a global data exchange. Fortunately, many if not most scientific applications exhibit this basic structure. Particle simulations are one such important class of applications. The particles could be atoms in a material science or biological simulation, stars in a simulation of galaxy formation, or units of charge in an electromagnetic application.

But by far the most common uses of graph partitioning involve computational meshes for the solution of differential equations. Finite volume, finite difference, and finite element methods all involve the decomposition of a complex geometry into simple shapes that interact only with near neighbors. Various graphs can be

constructed from the mesh and a partition of the graph identifies subregions to be assigned to processors. The numerical methods associated with such approaches are often very amenable to the graph partitioning approach. These ideas have been used to solve problems from many areas of computational science including fluid flow, structural mechanics, electromagnetics, and many more.

Graph Partitioning Algorithms for Parallel Computing

A wide variety of graph partitioning algorithms have been proposed for parallel computing applications. Here we review some of the more important approaches.

Geometric partitioning algorithms are very fast techniques for partitioning sets of entities that have an underlying geometry. For parallel computing applications involving simulations of physical phenomena in two or three dimensions, the corresponding data structures typically have geometric coordinates associated with each entity. Examples include molecules in atomistic simulations, masses in gravitational models, or mesh points in a finite element method. Recursive coordinate partitioning is a method in which the elements are recursively divided by planar cuts that are orthogonal to one of the axes [1]. This has the advantage of producing geometrically simple subdomains – just rectangular parallelepipeds. Recursive inertial bisection also uses planar cuts, but instead of being orthogonal to an axis, they are orthogonal to the direction of greatest inertia [9]. Intuitively, this is a direction in which the point set is elongated, so cutting perpendicular to this direction is likely to produce a smaller cut. Yet another alternative is to cut with circles or spheres instead of planes [8]. Geometric methods tend to be very fast, but produce low-quality partitions. They can be improved via local refinement methods like the approach of Fiduccia-Mattheyses discussed below.

A quite different set of approaches uses eigenvectors of a matrix associated with the graph. The most popular method in this class uses the second-smallest eigenvector of the Laplacian matrix of the graph [9]. A justification for this approach is beyond the scope of this article, but spectral methods generally produce partitions of fairly high quality. In a global sense, they find attractive regions for cutting a graph, but they are often poor in the fine details. This can be rectified by the application of a local refinement method. The main drawback of spectral methods is their high computational cost.

Local refinement methods are epitomized by the approach proposed by Fiduccia and Mattheyses [5] (FM). This method works by iteratively moving vertices between partitions in a manner that maximally reduces the size of the set of cut edges. Moves are considered even if they make the cut size larger since they may enable subsequent moves that lead to even better partitions. Thus, this method has a limited ability to escape from local minima to search for even better solutions. The key advance underlying FM is the use of clever data structures that allow all the moves and their consequences to be explored and updated efficiently. The FM algorithm is quite fast, and consistently improves results generated by other approaches. But since it only explores sets of partitions that are not far from the initial one, it is generally limited to making small changes and will not find better partitions that are quite different.

The most widely used class of graph partitioning techniques are multilevel algorithms as they provide a good balance between speed and quality. They were independently invented by several research groups more or less simultaneously [2, 4, 7]. Multilevel algorithms work by applying a local refinement method like FM at multiple scales. This largely overcomes the myopia that limits the effectiveness of local methods. This is accomplished by constructing a series of smaller and smaller graphs that roughly approximate the original graph. The most common way to do this is to merge small clusters of vertices within the original graph (e.g., combine two vertices sharing an edge into a single vertex). Once this series of graphs is constructed, the smallest graph is partitioned using any global method. Then the partition is refined locally and extended to the next larger graph. The refinement/extension process is repeated on larger and larger graphs until a partitioning of the original graph has been produced.

A number of general purpose global optimization approaches have been proposed for graph partitioning including simulated annealing, genetic algorithms, and tabu search. These methods can produce high-quality partitions but are usually very expensive and so are limited to niche applications within parallel computing.

Graph partitioning is often used as a preprocessing step to set up a parallel computation. The output of a

graph partitioner determines which objects are assigned to which processors, and appropriate input files and data structures are prepared for a parallel run. However, there are several situations in which the partitioning must be done in parallel. For a very large problem, the memory of a serial machine may be insufficient to hold the graph that needs to be partitioned. Also, for some classes of applications the structure of the computation changes over time. Examples include adaptive mesh simulations, or particle methods in which the particles move significantly. For such problems the work load must be periodically redistributed across the processors, and a parallel partitioning tool is required. Simple geometric algorithms have the advantage of being easy to parallelize, but multilevel partitioners have also been parallelized to provide higher-quality solutions. Techniques for effectively parallelizing such methods is an ongoing area of research.

A variety of open source graph partitioning tools have been developed in serial or parallel including Chaco, METIS, Jostle, and SCOTCH. Several of these are discussed in companion articles.

Limitations of Graph Partitioning

Although widely used to enable the parallelization of scientific applications, graph partitioning is an imperfect abstraction. For a parallel application to perform well, the work must be evenly distributed among processors and the cost of interprocessor communication must be minimized. Graph partition provides only a crude approximation for achieving these objectives.

In the graph partitioning model, each vertex is assigned a weight that is supposed to represent the time required to perform a piece of computation. On modern processors with complex memory hierarchies it is very difficult to accurately predict the runtime of a piece of code a priori. Cache performance can dominate runtime, and this is very hard to predict in advance. So the weights assigned to vertices in the graph partitioning model are just rough approximations.

An even more significant shortcoming of graph partitioning has to do with communication. For most applications, a vertex has data that needs to be known by all of its neighbors. If two of those neighbors are owned by the same processor, then that data need only be communicated once. In the graph partitioning model, two edges would be cut and so the actual volume of communication would be over-counted. Several alternatives to standard graph partitioning have been proposed to address this problem. In one approach, the number of vertices with off-processor neighbors is counted instead of the number of edges cut. A more powerful and elegant alternative uses hypergraphs and is sketched below.

Yet another deficiency in graph partitioning is that it emphasizes the total volume of communication. In many practical situations, latency is the performance-limiting factor, so it is the number of messages that matters most, not the size of messages.

As discussed above, graph partitioning is most appropriate for bulk synchronous applications. If the calculation involves complex interleaving of computations with communication or partial synchronizations then graph partitioning is less useful. An important application with this character is the factorization of a sparse matrix.

Finally, graph partitioning is only appropriate for applications in which the work and communication pattern are predictable and stable. This happens to be the case for many important scientific computing kernels, but there are other applications that do not fit this model.

Hypergraph Partitioning

A hypergraph is a generalization of a graph. Whereas a graph edge connects exactly two vertices, a *hyperedge* can connect any subset of vertices. This seemingly simple generalization leads to improved and more general partitioning models for parallel computing [3].

Consider a graph in which vertices represent computation and edges represent data dependencies. For each vertex, replace all the edges connected to it with a single hyperedge that joins the vertex and all of its neighbors. When the vertices are partitioned, if a particular vertex is separated from any of its neighbors, the corresponding hyperedge will be cut. For the common situation in which the vertex needs to communicate the same information to all of its neighbors, this single hyperedge will reflect the amount of data that needs to be shared with another processor. Thus, the number of cut hyperedges (or more generally the total weight of cut hyperedges) correctly captures the total volume of communication induced by a partitioning. In this way, the

hypergraph model resolves an important shortcoming of standard graph partitioning.

Hypergraphs also address a second deficiency of the graph model. If the communication is not symmetric (e.g., vertex *i* needs to send data to *j*, but *j* does not need to send data to *i*), then the graph model has difficulty capturing the communication requirements. The hypergraph model does not have this problem. A hyperedge simply spans a vertex *i* and every other vertex that *i* needs to send data to. There is no implicit assumption of symmetry in the construction of the hypergraph model.

Graph and hypergraph partitioning models, algorithms, and software continue to be active areas of research in parallel computing. PaToH and hMETIS are widely used hypergraph partitioning tools for parallel computing.

Related Entries

►Chaco
►Load Balancing, Distributed Memory
►Hypergraph Partitioning
►METIS and ParMETIS

Bibliographic Notes and Further Reading

Graph partitioning is a well-studied problem in theoretical computer science and is known to be difficult to solve optimally. For parallel computing, the challenge is to find algorithms that are effective in practice. Algebraic methods like Laplacian partitioning [9] are an important class of techniques, but can be expensive. Local refinement techniques like FM are also important [5], but get caught in local optima. Multilevel methods seem to offer the best trade off between cost and performance [2, 4, 7].

Hypergraph partitioning provides an important alternative to graph partition in many instances [3]. A survey of different partitioning models can be found in the paper by Hendrickson and Kolda [6].

Several good codes for graph partitioning are available on the Internet including Chaco, METIS, PaToH, and Scotch.

Acknowledgment

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the US Department of Energy under contract DE-AC04-94AL85000.

Bibliography

1. Berger MJ, Bokhari SH (1987) A partitioning strategy for nonuniform problems on multiprocessors. IEEE Trans Comput C-36(5):570–580
2. Bui T, Jones C (1993) A heuristic for reducing fill in sparse matrix factorization. In: Proceedings of the 6th SIAM Conference on parallel processing for scientific computing, SIAM, Portsmouth, Virginia, pp 445–452
3. Çatalyürek U, Aykanat C (1996) Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In: Lecture notes in computer science 1117, Proceedings Irregular'96, Springer-Verlag, Heidelberg, pp 75–86
4. Cong J, Smith ML (1993) A parallel bottom-up clustering algorithm with application to circuit partitioning in VLSI design. In: Proceedings of the 30th Annual CAM/IEEE International Design Automation Conference, DAC'93, ACM, San Diego, CA, pp 755–760
5. Fiduccia CM, Mattheyses RM (1982) A linear time heuristic for improving network partitions. In: Proceedings of the 19th ACM/IEEE Design Automation Conference, ACM/IEEE, Las Vegas, NV, June 1982, pp 175–181
6. Hendrickson B, Kolda T (2000) Graph partitioning models for parallel computing. Parallel Comput 26:1519–1534
7. Hendrickson B, Leland R (1995) A multilevel algorithm for partitioning graphs. In: Proceedings of Supercomputing '95, ACM, New York, December 1995. Previous version published as Sandia Technical Report SAND93-1301, Albuquerque, NM
8. Miller GL, Teng SH, Vavasis SA (1991) A unified geometric approach to graph separators. In: Proceedings of the 32nd Symposium on Foundations of Computer Science, IEEE, Pittsburgh, PA, October 1991, pp 538–547
9. Simon HD (1991) Partitioning of unstructured problems for parallel processing. In: Proceedings of the conference on parallel methods on large scale structural analysis and physics applications. Pergammon Press, Elmsford, NY

Graph Partitioning Software

►Chaco
►METIS and ParMETIS
►PaToH (Partitioning Tool for Hypergraphs)

Graphics Processing Unit

►NVIDIA GPU

Green Flash: Climate Machine (LBNL)

John Shalf[1], David Donofrio[1], Chris Rowen[2], Leonid Oliker[1], Michael Wehner[1]
[1]Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]CEO, Tensilica, Santa Clara, CA, USA

Synonyms

LBNL climate computer; Manycore; Tensilica; View from Berkeley

Definition

Green Flash is a research project focused on an application-driven manycore chip design that leverages commodity-embedded circuit designs and hardware/software codesign processes to create a highly programmable and energy-efficient HPC design. The project demonstrates how a multidisciplinary hardware/software codesign process that facilitates close interactions between applications scientists, computer scientists, and hardware engineers can be used to develop a system tailored for the requirements of scientific computing. By leveraging the efficiency gained from application-driven design philosophy, advanced processor synthesis tools from Tensilica, FPGA-accelerated architectural simulation from RAMP, auto-tuning for rapid optimization of the software implementation, the project demonstrated how a hardware/software codesign process can achieve a 100× increase in energy efficiency over its contemporaries using cost-effective commodity-embedded building blocks. To demonstrate application-driven design process, Green Flash was tailored for high-resolution global cloud resolving models, which are the leading justification for exascale computing systems. However, the approach can be generalized to a broader array of scientific applications. As such, Green Flash represents a vision of a new design process that could be used to develop effective exascale-class HPC systems.

Discussion

Introduction

The scientific community is facing one of its greatest challenges in the prediction of global climate change – a question whose answer has staggering economic, political, and sociological ramifications. The computational power required to inform such critical policy decisions requires a new breed of extreme scale computers to accurately model the global climate. The "business as usual" approach of using commercial off-the-shelf (COTS) hardware to build ever-larger clusters is increasingly unsustainable beyond the petaflop scale due to the constraints of power and cooling. Some estimates indicate an exaflop-capable machine would consume close to 180 MW of power. Such unreasonable power costs drive the need for a radically new approach to HPC system design. *Green Flash* is a theoretical system designed with an application-driven hardware and software codesign for HPC systems that leverages the innovative and low-power architectures and design processes of the low-power/embedded computing industry. Green Flash is the result of Berkeley Lab's research into energy-efficient system design – many details that are common to all system design, such as power, cooling, mechanical design, etc. are not addressed in this research as they are not unique to Green Flash and are challenges that would need to be overcome regardless of system architecture. The work presented here represents the energy efficiency gained through application-tailored architectures that leverage embedded processors to build energy efficient manycore processors.

History

In 2004, a group of University of California researchers, with backgrounds ranging from circuit design, computer architecture, CAD, embedded hardware/software, programming languages, compilers, applied math, to HPC, met for a period of two years to consider how current constraints on device physics at the silicon level would affect CPU design, system architecture, and programming models for future systems. The results of the discussions are documented in the University of California Berkeley Technical Report entitled "The Landscape of Parallel Computing Research: A View from Berkeley." This report was the genesis of the UC Berkeley ParLab, which was funded by Intel and Microsoft as well as the Green Flash project. Whereas the ParLab carried the work of the View from Berkeley forward for desktop and handheld applications,

the Green Flash project took the same principles and applied them to the design of energy-efficient scientific computing systems.

Hardware/software codesign, a methodology that allows both software optimization and semi-specialized processor design to be simultaneously developed, has long been a feature of power-sensitive embedded system designs, but thus far has seen very little application in the HPC space. However, given power has become the leading design constraint of future HPC systems and codesign, and other application-driven design processes have received considerably more attention. Green Flash leverages tools that were developed by Tensilica for rapid-synthesis of application-optimized CPU designs, and retargets them to designing processors that are optimized for scientific applications. The project also created novel inter-processor communication to enable that easier-to-program environment than its GPU contemporaries – providing hardware support for more natural programming environments based on partitioned global address space programming models.

Approach

It is widely agreed that architectural specialization can significantly improve efficiency, however, creating full-custom designs of HPC systems has often proven impractical due to excessive design/verification costs and lead-times. The embedded processor market relies on architectural customization to meet the demanding cost and power efficiency requirements of its products with a short turn-around time. With time to market a key element in profitability, sophisticated toolchains have been developed to enable rapid and cost-effective turn-around of power-efficient semicustom designs implementations appropriate to each specific processor design. Green Flash leverages these same toolchains to design power-efficient exascale systems, tailoring embedded chips to target scientific applications and providing a feedback path from the application programmer to the hardware design enabling a tight hardware/software codesign loop that is unprecedented in the HPC industry. *Auto-tuning* technologies are used to automate the software tuning process and maintain portability across the differing architectures produced inside the codesign loop. Auto-tuners can automatically search over a broad parameter space of optimizations to improve the computational efficiency of application kernels and help produce a more balanced architecture. To enable fast, accurate performance evaluation a Field Programmable Gate Array (FPGA) based hardware emulation platform will be used to allow an experimental architecture to be evaluated at speeds $1,000\times$ faster than typical software-based simulation methods – fast enough to allow execution of a real application rather than an arbitrary benchmark.

High-resolution global cloud system resolving models are the target application that will motivate Green Flash's architectural decisions. A truly exascale problem, a 1.5 km scale model would decompose the earth's atmosphere into twenty-billion individual cells and a machine with unprecedented performance would need to be realized in order for the model to run faster than real time. While using more power-efficient, off-the-shelf, embedded processors is a crucial first in meeting this challenge it is still insufficient. Green Flash will offer many other novel optimizations, both hardware and software, including alternatives to cache coherence that enable far more efficient inter-processor communication than a conventional symmetric multiprocessing (SMP) approach and aggressive, architecture-specific software optimization through auto-tuning. All these specialization techniques will allow Green Flash to efficiently meet the exascale computation requirements of global climate change prediction.

Modeling the Earth's Climate System

Current generation climate models are comprehensive representations of the various systems that determine the Earth's climate. Models prepared for the fourth report of the Intergovernmental Panel on Climate Change coupled submodels of the atmosphere, ocean, and sea ice together to provide simulations of the past, present, and future climate. It is expected that the major remaining components of the climate system, the terrestrial and oceanic biosphere, the Greenland and Antarctic ice sheets, and certain aspects of atmospheric chemistry will be represented in models currently being prepared for the next report. Each of the subsystem models has their own strengths and weaknesses, and each introduces a certain amount of uncertainty into projections of the future. Current computational resources limit the resolution of these submodels and are a contributor to these uncertainties. In

particular, resolution constraints on models of atmospheric processes do not allow clouds to be resolved forcing model developers to rely on sub-grid scale parameterizations based on statistical methods. However, simulations of the recent past produce cloud distributions that do not agree well with observations. These disagreements, traceable to the cumulus convection parameterizations, lead to other errors in patterns of the Earth's radiation and moisture budgets. Current global atmospheric models have resolutions of order 200 km, obviously many times larger than individual clouds. Development of models at the limit of the validity of cumulus parameterization (~25 km) is now underway by a few groups, although the necessary century scale integrations are just barely feasible on the largest current computing platforms. It is expected that many issues will be rectified by this increase in horizontal fidelity but that the fundamental limitations of cumulus parameterization will remain. The solution to this problem is to directly simulate cloud processes rather than attempt to model them statistically. At horizontal grid spacing of order ~ 1 km, cloud systems can be individually resolved providing this direct numerical simulation. However, the computation burden of fluid dynamics algorithms scales nonlinearly with the number of grid points due to time step limitations imposed by numerical stability requirements. Hence, the computational resources necessary to carry out century scale simulations of the Earth's climate dwarfs any traditional machine currently under development.

Climate Model Requirements

Extrapolation from measured computational requirements of existing atmospheric models allow estimates of what would be necessary at resolutions of order 1 km to support Global Cloud Resolving Models. To better make these estimates, the Green Flash project has partnered with Prof. David Randall's group at the Colorado State University (CSU). In their approach, the globe is represented by a mesh based on an icosahedron as the starting point. By successively bisecting the sides of the triangles making up this object, a remarkably uniform mesh on the sphere can be generated. However, this is not the only way to discretize the globe at this resolution and it will be important to have a variety of independent cloud system–resolving models if projections of the future are to have any credibility. For this reason it is important to emphasize that Green Flash will not be built to only run this particular discretization. Rather, this approach calls for optimizing a system for a class of scientific applications; therefore, Green Flash will be able to efficiently run most global climate models.

Extrapolation based on today's cluster-based, general purpose, HPC systems produce estimates that the *sustained* computational rate necessary to simulate the Earth's climate 1,000 times faster than it actually occurs was 10 Pflops. A tentative estimate from the CSU model is as much as 70 Pflops. This difference can be regarded as one measure of the considerable uncertainty in making these estimates. As the CSU model matures, there will be the opportunity to determine this rate much more accurately. Multiple realizations of individual simulations are necessary to address the statistical complexities of climate system. Hence, an exaflop scale machine would be necessary to carry out this kind of science. The exact peak flop rate required depends greatly on the efficiency that the machine could be used.

These enormous sustained computational rates are not even imaginable if there is not enough parallelism in the climate problem. Fortunately, cloud system resolving models at the kilometer scale do offer plenty of opportunity to decompose the physical domain. Bisection of the triangles composing the icosahedron twelve successive times produces a global mesh with 167,772,162 vertices spaced between 1 and 2 km apart. A logically rectangular two-dimensional domain decomposition strategy can be applied horizontally to the icosahedral grid. Choosing square segments of the mesh containing 64 grid points each (8 × 8) results in 2,621,440 horizontal domains. The vertical dimension offers additional parallelism. Assuming that 128 layers could be decomposed in 8 separate vertical domains, the total number of physical sub-domains could be 20,971,520.

Twenty-one million way parallelism may seem mind-boggling but this particular strawman decomposition was devised with practical constraints on the performance of an individual core in an SMP in mind. Each of the 21-million cores in this system will be assigned a small group of sub-domains on which they will execute the full (physics, dynamics, etc.) climate model. Flops per watt is the key performance metric for designing the SMP for Green Flash, and the goal of 100× energy efficiency over existing machines will be achieved by

tailoring the architecture to the needs of the climate model. One example that drives the need for a high core count per socket is the model's communication pattern. While the model is nearest-neighbor dominated, the majority of the more latency-sensitive communication occurs between the vertical layers. By keeping this communication on-chip the more latency tolerant horizontal communication can be sent off-chip with less performance penalty. Looking at per-core features, by running with a lower (500 MHz) clock speed relative to today's server-class processors Green Flash gains a cubic improvement in power consumption. Removal of processor features that are nonoptimal for science such as Translation Look-aside Buffers (TLBs) and Out of Order (OOO) processing creates a smaller die, which reduces leakage to help further reduce power.

Hardware Design Flow

Verification costs are quickly becoming the dominant force in custom hardware solutions. Large arrays of simple processors again hold a significant advantage here, as the work to verify to a simple processing element and then replicate them on die is significantly lower. The long lead times and high costs of doing custom designs has generally dissuaded the HPC community from custom solutions and pushed more for clusters of COTS (Commercial off-the-self) hardware. The traditional definition of COTS in the HPC space is typically at the board or socket level; Green Flash seeks to redefine this notion of COTS and asserts that a custom processor made up of pre-verified building blocks can still be considered COTS hardware. This fine grained view permits Green Flash to benefit from both the architectural specialization afforded by these specialized processing elements and the shorter lead times and reduced verification costs that come with using a building block approach.

The constraints of power have long directed the development of embedded architectures and so it is advantageous to begin with an embedded core and leverage the sophisticated tool chains developed to minimize time from architectural specifications to ASIC. These toolchains start with a collection of pre-verified functional units and allow them to be combined in a myriad of ways rapidly producing power-efficient semi-custom designs. For instance, starting with a base architecture a designer may wish to add floating-point support to a processor, or perhaps add a larger cache or local store. These functional units can be added to a processor design as easily as clicking a checkbox or dropdown menu. The tool will then select the correct functional unit from its library and integrate it into the design – all without the designer needing to intervene. These tools eliminate large amounts of not only boilerplate, but also full custom logic that once needed to be written and re-written in order to change a processor's architecture. Of course the tools are not boundless and are subject to the same design limitations as any other physical design process – for instance, one cannot efficiently have hundreds of read ports from a single memory, but the amount of flexibility created through these tools vastly outweighs any inherent limitations.

The rapid generation of processor cores alone makes these tools very interesting, however, the overhead of generating a usable software stack for each processor would negate the time saved developing the hardware. While adding caches or changing bus widths has little effect on the ISA, and therefore a minimal software impact, adding a new functional unit such as floating-point or integer division has a large impact on the software flow. Building custom hardware creates significant work not only in the creation of a potentially complex software stack but also a time-consuming verification process. As with the software stack, without a method to jump-start the verification process the tools would begin to lose their effectiveness. To address both of these critical issues these tools generate optimizing compilers, test benches as well as a functional simulator in parallel with the RTL for the design. Having the processor constructed of pre-verified building blocks combined with the automatic generation of test benches greatly reduces the risk and time required for formal verification. To help maintain backward and general purpose compatibility the processor's ISA is restricted to one that is functionally complete and allows for the execution of general purpose code.

A Science Optimized Processor Design

The processor design for Green Flash is driven by efficiency and the best way to reduce power consumption and increase efficiency is to reduce waste. With that in mind, the target architecture calls for a very simple, in order core with no branch prediction. The heavy memory and communication requirements demanded by the

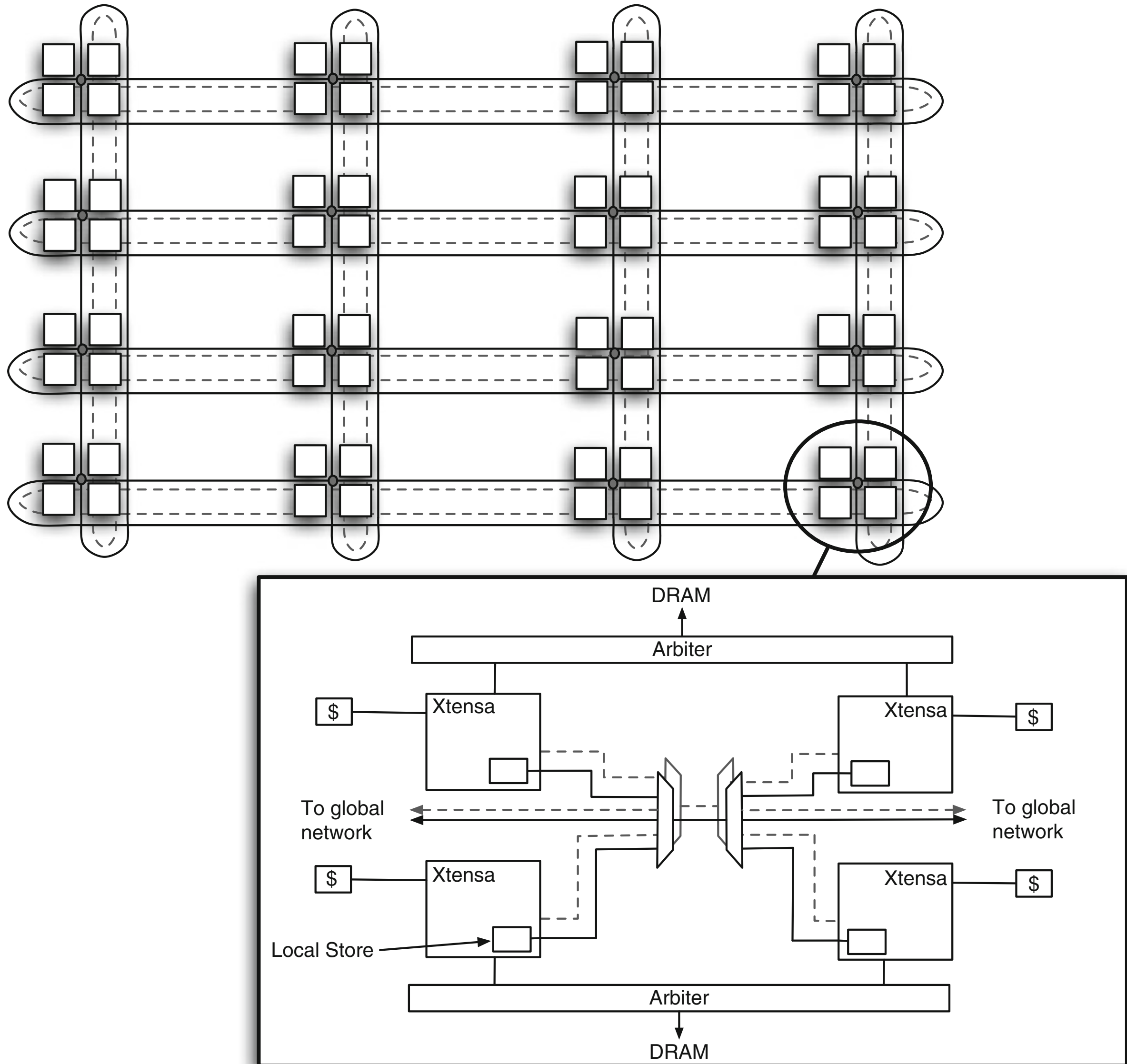
climate model have imparted the greatest influence on the design of the Green Flash core. Building on prior work from Williams et al. where it was shown that for memory-intensive applications, cores with a local store, such as Cell, were able to utilize a higher percentage of the available DRAM bandwidth, the target processor architecture includes a local store. In the Green Flash SMP design there will be two on-chip networks – as illustrated in Fig. 1. As can be somewhat expected, the majority of communication that occurs between subdomains within the climate model is nearest neighbor. Building on work from both Balfour and Dally [11] a packet switched network with a concentrated torus topology was chosen for the SMP as it has been shown to provide superior performance and energy efficiency for codes where the dominant communication pattern is nearest neighbor.

To further optimize the Green Flash processor for science the programming model is being considered a first class citizen when designing the architecture. Traditional cache coherent models found in many modern SMPs do not allow fine-grained synchronization between cores. In fact, when benchmarking the current climate model on present day machines it is shown that greater than 90% of execution time is spent in communication. By creating an architecture where an individual core will not pay a huge overhead penalty for sending or receiving a relatively small message the amount of time spent in communication can be greatly reduced. The processing cores used in the Green Flash SMP have powerful, flexible streaming interfaces. Each processor can have multiple, designer defined ports with a simple FIFO-like interface with each port capable of sending and receiving a packet of data on each clock. This low-overhead streaming interface will bypass the cache and connect to one of the torus networks on chip. This narrow network can be used for exchange of addresses, while the wider torus network is used for exchange of data. Following a Partitioned Global Address Space (PGAS) model the address space for each processor's local store is mapped into the global address space and the data exchange is done as a DMA from local store to local store. This allows the communication between processors to map very well to a MPI send/receive model used by the climate model and many other scientific codes. The view to the programmer will be as though all processors are directly connected to their neighbors. To further simplify programming, a traditional cache hierarchy is also in place to allow codes to be slowly ported to the more efficient local-store based interprocessor network. In order to minimize power, the use of photonic interlinks for the inter-core network is being investigated as an efficient method of transferring long messages. In the case of Green Flash, the data network is one cache line in width and will consist of several phases per message.

Hardware/Software Codesign Strategy

Conventional approaches to hardware design generally have a long latency between hardware design and software development/optimization so designers frequently rely on benchmark codes to find a power-efficient architecture. However, modern compilers fail to generate even close to optimal code for target machines. Therefore, a benchmark-based approach to hardware design does not exploit the full performance potential of the architecture design points under consideration leading to possibly sub-optimal hardware solutions. The success of auto-tuners has shown that it is still possible to generate efficient code using domain knowledge. In combination with the ability to rapidly produce semi-custom hardware designs a tight, effective hardware/software codesign loop can be created. The codesign approach, as shown in Fig. 2 incorporates extensive software tuning into the process of hardware design. Hardware design space exploration is routinely done to tailor the hardware design parameters to the target applications. The auto-tuned software tailors the application to the hardware design point under consideration by empirically searching over a range of software implementations to find the best mapping of the software to the micro-architecture. One of the hindrances toward the practical relevance of codesign is the large hardware/software design space exploration. Conventional hardware design approaches use software simulation of hardware to perform hardware design space exploration. Because codesign involves searching over a much larger design space (there is now a need to explore the software design space at each hardware design point), codesign is impractical if software simulation of hardware is used.

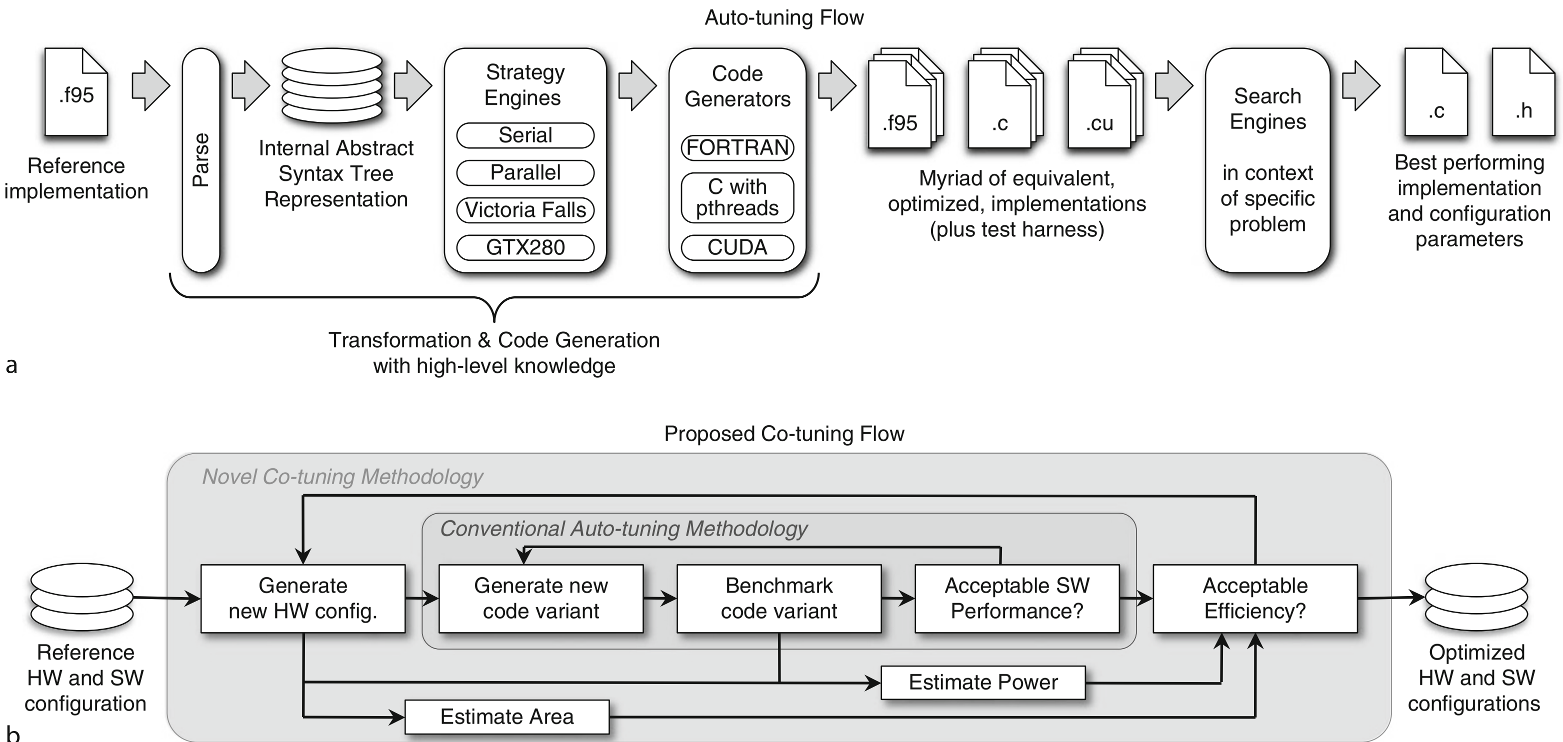
Rather than be constrained by the limitations of a software simulation environment, it is possible instead to take advantage of the processor generation



Green Flash: Climate Machine (LBNL). Fig. 1 A concentrated torus network fabric yields the highest performance and most power efficient design for scientific codes

toolchain's ability to create synthesizable RTL for any given processor. By loading this design onto an FPGA, a potential processor design can be emulated running 500× faster than a functional simulator. This speedup allows the benchmarking of true applications rather than being forced to rely on representative code snippets or statically defined benchmarks. Furthermore, this speed advantage does not come at the expense of accuracy; to the contrary, FPGA emulation is arguably much more accurate than a software simulation environment as it truly represents the hardware design. This fast accurate emulation environment provides the ability to run and benchmark the actual climate model as it is being developed and allows the codesign infrastructure to quickly search a large design space.

The speed and accuracy advantages of using FPGAs have typically been dwarfed by the increased complexity of coding in Verilog or VHDL versus C++ or Python



Green Flash: Climate Machine (LBNL). Fig. 2 The result of combining an existing auto-tuning framework (**a**) with a rapid hardware design cycle and FPGA emulation is (**b**) a proposed hardware/software codesign flow

as well as the ability to emulate large designs due to limitations in FPGA area/LUT count. The practicality of using FPGAs for large system emulation has increased dramatically over the past decade. The ability to access relatively large dynamic memories, such as DDR, has always been a difficult challenge with FPGAs due to the tight timing requirements. FPGA vendors, such as Xilinx, have eased this difficulty by providing IP through its Memory Interface Generator (MIG) tool and adding IO features to the Virtex-5 series. Freely available Verilog IP libraries – whether they are Xilinx CoreGen, or the RAMP group's GateLib – allow for a modular, building block approach to HW design. Finally, while commercial microprocessors are experiencing a plateau in their clock rates and power consumption, FPGAs are not. FPGA LUT count continues to increase allowing the emulation of more complex designs and FPGA clocks, while traditionally significantly slower than commercial microprocessor clock rates have been growing steadily, closing the gap between emulated and production clock rates. In the case of Green Flash, the relatively low target clock frequency (500 MHz) of the final ASIC is an additional motivation to target an FPGA emulation environment. The current emulated processor design runs at 33 MHz – a significant fraction of the target clock rate. This relatively high speed enables the efficient benchmarking of an entire application rather than a representative portion.

While the steady growth in LUT count on FPGAs has enabled the emulation of more complex designs, with a strawman architecture of 128 cores per socket it is necessary to emulate more than the two or four cores that will fit on a single FPGA. To scale beyond the cores that will fit on a single FPGA a multi-FPGA system, such as the Berkeley Emulation Engine (BEE3) can be used. The BEE3 board has four Virtex-5 155 FPGAs connected in a ring with a cross-over connection. Each FPGA has access to two channels of DDR2 memory allowing 4 GB of memory per FPGA. The BEE3 will allow effective emulation of eight cores with the appropriate NoC infrastructure per board. To scale beyond eight cores, the BEE3 includes 10 Gb connections allowing the boards to be linked and emulation of an entire socket becomes possible. There is significant precedence for emulation of massively multithreaded architectures across multiple FPGAs. One recent example was demonstrated by the Berkeley RAMP Blue project where over 1,000 cores were emulated using a stack of 16 BEE2 boards.

Hardware Support for New Programming Models

Applications and algorithms will need to rely increasingly on fine-grained parallelism and strong scaling and support fault resilience to accommodate the massive growth of explicit on-chip parallelism and constrained bandwidth anticipated for future chip architectures. History shows that the application-driven approach offers the most productive strategy for evaluating and selecting among the myriad choices for refactoring algorithms for full scientific application codes as the industry moves through this transitional phase. Green Flash functions as a testbed to explore novel programming models together with hardware support to express fine-grained parallelism to achieve performance, productivity, and correctness for leading-edge application codes in the face of massive parallelism and increasingly hierarchical hardware. The goal of this development thrust is to create a new software model that can provide a stable platform for software development for the next decade and beyond for all scales of scientific computing.

The Green Flash design created direct hardware support for both the message passing interface (MPI) and partitioned global address space (PGAS) programming models to enable scaling of these familiar single program, multiple data (SPMD) programming styles to much larger-scale systems. The modest hardware support enables relatively well-known programming paradigms to utilize massive on-chip concurrency and to use hierarchical parallelism to enable use of larger messages for interchip communication.

However, not all applications will be able to express parallelism through simple divide-and-conquer problem partitioning. So the message-queues and software-managed memories that are used to implement PGAS are also being used to explore new asymmetric and asynchronous approaches to achieving strong-scaling performance improvements from explicit parallelism. Techniques that resemble class static dataflow methods are garnering renewed interest because of their ability to flexibly schedule work and to accommodate state migration to correct load imbalances and failures. In the case of the climate code, dataflow techniques can be used to concurrently schedule the physics computations

with the dynamic core of the climate code, thereby doubling the effective concurrency without moving to a finer domain decomposition. This approach also benefits from the unique interprocessor communication interfaces developed for Green Flash.

Fault Resilience

A question that comes up when proposing a 20 million processor computing system is how to deal with fault resilience. While trying not to trivialize the issue, it should be noted that this is a problem for everyone designing large-scale machines. The proposed approach of using many simpler cores does not introduce any unique challenges that are different than the challenges faced for aggregating conventional server chips into large-scale systems provided the total number of discrete chips in the system is not dramatically different. The following observations are made to qualify this point.

For a given silicon process (e.g., a 65 nm process and same design rules)

1. Hard failure rates are primarily proportional to the number of sockets in a system (e.g., solder joint failures, weak wire bonds, and a variety of mechanical and electrical issues) and secondarily related to the total chip surface area (probability of defect vs tolerance of the design rules to process variation). It is not proportional to the number of processor cores per se given that the cores come in all shapes and sizes.
2. Soft error rates caused by comic rays roughly proportional to chip surface area when comparing circuits that employ the same process technology (e.g., 65 nm).
3. Bit error rates for data transfer tend to increase proportionally with clock rate.
4. Thermal stress is also a source of hard errors.

For hard errors:

1. Spare cores can be designed into each ASIC to tolerate defects due to process variation. This approach is already used by the 188 core Cisco Metro chip, which incorporates 8 spare cores (192 cores in total) to cover chip defects.
2. Each chip is expected to dissipate a relatively small 7–15 W (or that is the target) subjecting them to less mechanical/thermal stress.
3. It has been demonstrated that Green Flash can achieve more *delivered* performance out of fewer sockets, which reduces exposure to hard-failures due to bad electrical connections or other mechanical/electrical defects.
4. Like BlueGene, memory and CPUs can be flow-soldered onto the board to reduce hard and soft failure rates for electrical connections given removable sockets are far more susceptible to both kinds of faults. So eliminating removable sockets can greatly reduce error rates.

For soft errors:

1. All of the basics for reliability and error recovery in the memory subsystem including full ECC (error correcting code) protection for caches and memory interfaces are included in the design.
2. Using many simpler cores allows fewer sockets to be used and less silicon surface area to achieve the same delivered performance. So that is to say, Green Flash has less exposure to major sources of failure than a conventional high-frequency core design. Therefore, fewer sockets and fewer random bit-flips due to mechanical noise and other stochastic error sources.
3. The core clock frequency of 500 MHz improves Signal to Noise Ratio for on-chip data transfers.
4. Incorporation of a Nonvolatile Random Access Memory (NVRAM) memory controller and channel on each System on Chip (SoC). Each node can copy the image of memory to the NVRAM periodically to do local checkpoints. If there is a soft-error (e.g., an uncorrectable memory error), then the node can initiate a roll-back to the last available checkpoint. For hard failures (e.g., a node does and cannot be revived), the checkpoint image will be copied to neighboring nodes on a periodic basis to facilitate localized state recovery. Both strategies enable much faster roll-back when errors are encountered than the conventional user-space checkpointing approach.

Therefore, the required fault resilience strategies will bear similarity to other systems that employ a similar number of sockets (~120, 000), which are not unprecedented. The BlueGene system at Lawrence Livermore National Laboratory contains a comparable number of sockets, and achieves a 7–15 day Mean Time Between Failures (MTBF), which is far longer than systems that contain a fraction the number of processor cores. Therefore, careful application of well-known fault-resilience techniques together with a few

novel "extended fault resilience" mechanisms such as localized NVRAM checkpoints can achieve an acceptable MTBF for extreme-scale implementations of this approach to system design.

Conclusions

Green Flash proposes a radical approach of application-driven computing design to break through the slow pace of incremental changes, and foster a sustainable hardware/software ecosystem with broad-based support across the IT industry. Green Flash has enabled the exploration of practical advanced programming models together with lightweight hardware support mechanisms that allow programmers to utilize massive on-chip concurrency, thereby creating the market demand for massively concurrent components that can also be the building block of midrange and extreme-scale computing systems. New programming models must be part of a new software development ecosystem that spans all scales of systems, from midrange to the extreme-scale to facilitate a viable migration path from development to large-scale production computing systems. The use of the FPGA-based hardware emulation platforms, such as RAMP, to prototype and run hardware prototypes at near-realtime speeds before it is built allow testing of full-fledged application codes and advanced software development to commence many years before the final hardware platform is constructed. These tools have enabled a tightly coupled software/hardware codesign process that can be applied effectively to the complex HPC application space.

Rather than ask "what kind of scientific applications can run on our HPC cluster after it arrives," the question should be turned around to ask "what kind of system should be built to meet the needs of the most important science problems." This approach is able to realize its most substantial gains in energy-efficiency by peeling back the complexity of high-frequency microprocessor design point to reduce sources of waste (wasted opcodes, wasted bandwidth, waste caused by orienting architectures toward serial performance). BlueGene and SiCortex have demonstrated the advantages of using the simpler low-power embedded processing elements to create energy-efficient computing platforms. However, the Green Flash codesign approach goes beyond traditional embedded core design point of BlueGene and SiCortex by using explicit message queues and software controlled memories to further optimize data movement, while still retaining a smaller conventional cache-hierarchy only to support incremental porting to the more energy and bandwidth-efficient design point. Furthermore, simple hardware support for lightweight on-chip interprocessor synchronization and communication make it much simpler and straightforward and efficient to program massive arrays of processors than more exotic programming models such as CUDA and Streaming.

Green Flash has been a valuable research vehicle to understand how the evolution of massively parallel chip architectures can be guided by close-coupled feedback with the design of the application, algorithms, and hardware together. Application-driven design ensures hardware design decisions do not evolve in reaction to hardware constraints, without regard to programmability and delivered application performance. The design study has been driven by a deep dive into the climate application space, but enables explorations that cut across all application areas and have ramifications to the next generation of fully general-purpose architectures. Ultimately, Green Flash should consist of an architecture that can maximally leverage reusable components from the mass market of the embedded space while improving the programmability for the many-core design point. The building blocks of a future HPC system must be the preferred solution in terms of performance and programmability for everything from the smallest high-performance energy-efficient embedded system, to midrange departmental systems, to the largest-scale systems.

Bibliography

1. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA (2006) The landscape of parallel computing research: a view from Berkeley. Technical report no. UCB/EECS-2006-183, EECS Department University of California, Berkeley
2. Wehner M, Oliker L, Shalf J (2008) Towards ultra-high resolution models of climate and weather. Int J High Perform Comput Appl 22:149–165
3. Shalf J (2007) The new landscape of parallel computer architecture. J Phys: Conf Ser 78:012066
4. Donofrio D, Oliker L, Shalf J, Wehner MF, Rowen C, Krueger J, Kamil S, Mohiyuddin M (2009) Energy-efficient computing for extreme-scale science. IEEE Computer, Los Almitos
5. Wehner M, Oliker L, Shalf J (2009) Low-power supercomputers. IEEE Spectrum 29(9):66–68
6. Shalf J, Wehner M, Oliker L, Hules J (2009) Green flash project: the challenge of energy-efficient HPC. SciDAC Review, Fall

7. Kamil SA, Shalf J, Oliker L, Skinner D (2005) Understanding ultra-scale application communication requirements. In: IEEE international symposium on workload characterization (IISWC) Austin, 6–8 Oct 2005 (LBNL-58059)
8. Kamil S, Chan Cy, Oliker L, Shalf J, Williams S (2010) An auto-tuning framework for parallel multicore stencil computations. In: IPDPS 2010, Atlanta
9. Hendry G, Kamil SA, Biberman A, Chan J, Lee BG, Mohiyuddin M, Jain A., Bergman K, Carloni LP, Kubiatocics J, Oliker L, Shalf J (2009) Analysis of photonic networks for chip multiprocessor using scientific applications. In: NOCS 2009, San Diego
10. Mohiyuddin M, Murphy M, Oliker L, Shalf J, Wawrzynek J, Williams S (2009) A design methodology for domain-optimized power-efficient supercomputing, In: SC 09, Portland
11. Balfour J, Dally WJ (2006) Design tradeoffs for tiled cmp on-chip networks. In ICS '06: Proceedings of the 20th annual international conference on supercomputing

Grid Partitioning

►Domain Decomposition

Gridlock

►Deadlocks

Group Communication

►Collective Communication

Gustafson's Law

John L. Gustafson
Intel Labs, Santa Clara, CA, USA

Synonyms

Gustafson–Barsis Law; Scaled speedup; Weak scaling

Definition

Gustafson's Law says that if you apply P processors to a task that has serial fraction f, scaling the task to take the same amount of time as before, the speedup is

$$\begin{aligned}\text{Speedup} &= f + P(1-f)\\ &= P - f(P-1)\,.\end{aligned}$$

It shows more generally that the serial fraction does not theoretically limit parallel speed enhancement, if the problem or workload scales in its parallel component. It models a different situation from that of Amdahl's Law, which predicts time reduction for a fixed problem size.

Discussion

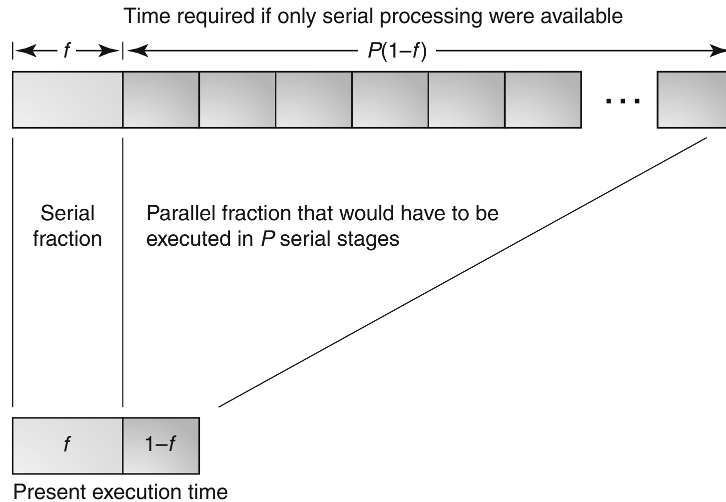
Graphical Explanation

Figure 1 explains the formula in the Definition:

The time the user is willing to wait to solve the workload is unity (lower bar). The part of the work that is observably serial, f, is unaffected by parallelization. The remaining fraction of the work, $1 - f$, parallelizes perfectly so that a serial processor would take P times longer to execute it. The ratio of the top bar to the bottom bar is thus $f + P(1 - f)$. Some prefer to rearrange this algebraically as $P - f(P - 1)$.

The diagram resembles the one used in the explanation of Amdahl's Law (see ►Amdahl's Law) except that Amdahl's Law fixes the *problem size* and answers the question of how parallel processing can reduce the execution time. Gustafson's Law fixes the *run time* and answers the question of how much longer time the present workload would take in the absence of parallelism [5]. In both cases, f is the experimentally observable fraction of the current workload that is serial. The similarity of the diagram to the one that explains Amdahl's Law has led some to "unify" the two laws by a change of variable. It is an easy algebraic exercise to set the upper bar to unit time and express the f of Gustafson's Law in terms of the variables of Amdahl's Law, but this misses the point that the two laws proceed from *different premises*. Every attempt at unification begins by applying the same premise, resulting in a circular argument that the two laws are the same.

The fundamental underlying observation of Gustafson's Law is that more powerful computer systems usually solve larger problems, not the same size problem in less time. Hence, a performance enhancement like parallel processing expands what a user can do with a computing system to match the time the user is willing to wait for the answer. While computing power has increased by many orders of magnitude over the last



Gustafson's Law. Fig. 1 Graphical derivation of Gustafson's Law

half-century (see ▸Moore's Law), the execution time for problems of interest has been constant, since that time is tied to human timescales.

History

In a 1967 conference debate over the merits of parallel computing, IBM's Gene Amdahl argued that a considerable fraction of the work of computers was inherently serial, from both algorithmic and architectural sources. He estimated the serial fraction f at about 0.25–0.45. He asserted that this would sharply limit the approach of parallel processing for reducing execution time [1]. Amdahl argued that even the use of two processors was less cost-effective than a serial processor. Furthermore, the use of a large number of processors would never reduce execution time by more than $1/f$, which by his estimate was a factor of about 2–4.

Despite many efforts to find a flaw in Amdahl's argument, "Amdahl's Law" held for over 20 years as justification for the continued use of serial computing hardware and serial programming models.

The Rise of Microprocessor-Based Systems

By the late 1970s, microprocessors and dynamic random-access memory (DRAM) had dropped in price to the point where academic researchers could afford them as components in experimental parallel designs. Work in 1983 by Charles Seitz at Caltech using a message-passing collection of 64 microprocessors [11] showed excellent absolute performance in terms of floating-point operations per second, and seemed to defy Amdahl's pessimistic prediction. Seitz's success led John Gustafson at FPS to drive development of a massively parallel cluster product with backing from the Defense Advanced Research Projects Agency (DARPA). Although the largest configuration actually sold of that product (the FPS T Series) had only 256 processors, the architecture permitted scaling to 16,384 processors. The large number of processors led many to question: *What about Amdahl's Law?* Gustafson formulated a counterargument in April 1986, which showed that performance is a function of both the problem size and the number of processors, and thus Amdahl's Law need not limit performance. That is, the serial fraction f is not a constant but actually decreases with increased problem size. With no experimental evidence to demonstrate the idea, the counterargument had little impact on the computing community.

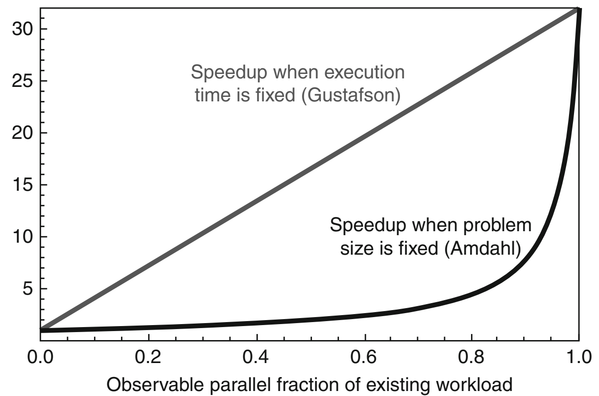
An idea for a source of experimental evidence arose in the form of a challenge that Alan Karp had publicized the year before [8]. Karp had seen announcements of the 65,536-processor CM-1 from Thinking Machines and the 1,024-processor NCUBE10 from nCUBE, and believed Amdahl's Law made it unlikely that such massively parallel computers would achieve a large fraction of their rated performance. He published a skeptical challenge and a financial reward for anyone who could demonstrate a parallel speedup of over 200 times on

three real applications. Karp suggested computational fluid dynamics, structural analysis, and econometric modeling as the three application areas and gave some ground rules to insure that entries focused on honest parallel speedup without tricks or workarounds. For example, one could not cripple the serial version to make it artificially 200 times slower than the parallel system. And the applications, like the three suggested, had to be ones that had interprocessor communication throughout their execution as opposed to "embarrassingly parallel" problems that had communication only at the beginning and end of a run.

By 1987, no one had met Karp's challenge, so Gordon Bell adopted the same set of rules and suggested applications as the basis for the Gordon Bell Award, softening the goal from 200 times to whatever the best speedup developers could demonstrate. Bell expected the initial entries to achieve about tenfold speedup [2].

The purchase by Sandia National Laboratories of the first 1,024-processor NCUBE 10 system created the opportunity for Gustafson to demonstrate his argument on the experiment outlined by Karp and Bell, so he joined Sandia and worked with researchers Gary Montry and Robert Benner to demonstrate the practicality of high parallel speedup. Sandia had real applications in fluid dynamics and structural mechanics, but none in econometric modeling, so the three researchers substituted a wave propagation application. With a few weeks of tuning and optimization, all three applications were running at over 500-fold speedup with the fixed-size Amdahl restriction, and over 1,000-fold speedup with the scaled model proposed by Gustafson. Gustafson described his model to Sandia Director Edwin Barsis, who suggested explaining scaled speedup using a graph like that shown in Fig. 2.

Barsis also insisted that Gustafson publish this concept, and is probably the first person to refer to it as "Gustafson's Law." With the large experimental speedups combined with the alternative model, *Communications of the ACM* published the results in May 1988 [5]. Since Gustafson credited Barsis with the idea of expressing the scaled speedup model as graphed in Fig. 2, some refer to Gustafson's Law as the Gustafson–Barsis Law. The three Sandia researchers



Gustafson's Law. Fig. 2 Speedup possible with 32 processors, by Gustafson's Law and Amdahl's Law

published the detailed explanation of the application parallelizations in a *Society of Industrial and Applied Mathematics* (SIAM) journal [4].

Parallel Computing Watershed

Sandia's announcement of 1,000-fold parallel speedups created a sensation that went well beyond the computing research community. Alan Karp announced that the Sandia results had met his Challenge, and Gordon Bell gave his first award to the three Sandia researchers. The results received publicity beyond that of the usual technical journals, appearing in *TIME*, *Newsweek*, and the US Congressional Record. Cray, IBM, Intel, and Digital Equipment began work in earnest developing commercial computers with massive amounts of parallelism for the first time.

The Sandia announcement also created considerable controversy in the computing community, partly because some journalists sensationalized it as a proof that Amdahl's Law was false or had been "broken." This was never the intent of Gustafson's observation. He maintained that Amdahl's Law was the correct answer but to the wrong question: "How much can parallel processing reduce the run time of a current workload?"

Observable Fraction and Scaling Models

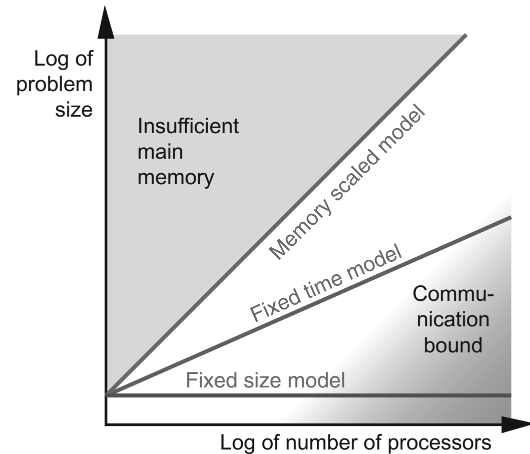
As part of the controversy, many maintained that Amdahl's Law was still the appropriate model to use in all situations, or that Gustafson's Law was simply

a corollary to Amdahl's Law. For scaled speedup, the argument went that one simply works backward to determine what the f fraction in Amdahl's Law must have been to yield such performance. This is an example of circular reasoning, since the proof that Amdahl's Law applies begins by assuming it applies.

For many programs, it is possible to instrument and measure the fraction of time f spent in serial execution. One can place timers in the program around serial regions and obtain an estimate of f. This fraction then allows Amdahl's Law estimates of time reduction, or Gustafson's Law estimates of scaled speedup. Neither law takes into account communication costs or intermediate degrees of parallelism. (When communication costs are included in Gustafson's fixed-time model, the speedup is again limited as the number of processors grows, because communication costs rise to the point where there is no way to increase the size of the amount of work without increasing the execution time.)

A more common practice is to measure the parallel speedup as the number of processors is varied, and fit the resulting curve to derive f. This approach confuses serial fraction with communication overhead, load imbalance, changes in the relative use of the memory hierarchy, and so on. Some refer to the requirement to keep the problem size the same yet use more processors as "strong scaling." Still, a common phenomenon that results from "strong scaling" is that it is *easier*, not *harder*, to obtain high amounts of speedup. When spreading a problem across more and more processors, the *memory per processor* goes down to the point where the data fits entirely in cache, resulting in *superlinear speedup* [3]. Sometimes, the superlinear speedup effects and the communication overheads partially cancel out, so what appears to be a low value of f is actually the result of the combination of the two effects. In modern parallel systems, performance analysis with either Amdahl's Law or Gustafson's Law will usually be inaccurate since communication costs and other parallel processing phenomena have large effects on the speedup.

In Fig. 3, Amdahl's Law governs the Fixed-Sized Model line, Gustafson's Law governs the Fixed-Time Model line, and what some call the Sun–Ni Law governs the Memory Scaled Model [12]. The fixed-time model line is an irregular curve in general, because of



Gustafson's Law. Fig. 3 Different scaling types and communication costs

the communication cost effects and because the percentage of the problem that is in each memory tier (mass storage, main RAM, levels of cache) changes with the use of more processors.

Analogies

There are many aspects of technology where an enhancement for time reduction actually turns out to be an enhancement for what one can accomplish in the same time as before. Just as Amdahl's Law is an expression of the more general Law of Diminishing Returns, Gustafson's Law is an expression of the more general observation that technological advances are used to improve what humans accomplish in the length of time they are accustomed to waiting, not to shorten the waiting time.

Commuting Time

As civilization has moved from walking to horses to mechanical transportation, the average speed of getting to and from work every day has gone up dramatically. Yet, people take about half an hour to get to or from work as a tolerable fraction of the day, and this amount of time is probably similar to what it has been for centuries. Cities that have been around for hundreds or thousands of years show a concentric pattern that reflect the increasing distance people could commute for the amount of time they were able to tolerate.

Transportation provides many analogies for Gustafson's Law that expose the fallacy of fixing the size of a problem as the control variable in discussing large performance gains. A commercial jet might be able to travel 500 miles per hour, yet if one asks "How much will it reduce the time it takes me presently to walk to work and back?" the answer would be that it does not help at all. It would be easy to apply an Amdahl-type argument to the time to travel to an airport as the serial fraction, such that the speedup of using a jet only applies to the remaining fraction of the time and thus is not worth doing. However, this does not mean that commercial jets are useless for transportation. It means that faster devices are for larger jobs, which in this case means longer trips.

Here is another transportation example: If one takes a trip at 30 miles per hour and immediately turns around, how fast does one have to go to average 60 miles per hour? This is a trick question that many people incorrectly answer, "90 miles per hour." To average 60 miles per hour, one would have to travel back at infinite speed, that is, return *instantly*. Amdahl's Law applies to this fixed-distance trip. However, suppose the question were posed differently: "If one travels for an hour at 30 miles per hour, how fast does one have to travel in the next hour to average 60 miles per hour?" In that case, the intuitive answer of "90 miles per hour" *is the correct one*. Gustafson's Law applies to this fixed-time trip.

The US Census

In the early debates about scaled speedup, Heath and Worley [6] provided an example of a fixed-sized problem that they said was not appropriate for Gustafson's Law and for which Amdahl's Law should be applied: the US Census. While counting the number of people in the USA would appear to be a fixed-sized problem, it is actually a perfect example of a fixed-time problem since the Constitution mandates a complete headcount every 10 years. It was in the late nineteenth century, when Hollerith estimated that the population had grown to the point where existing approaches would take longer than 10 years that he developed the card punch tabulation methods that made the process fast enough to fit the fixed-time budget.

With much faster computing methods now available, the Census process has grown to take into account many more details about people than the simple head count that the Constitution mandates. This illustrates a connection between Gustafson's Law and the jocular Parkinson's Law: "Work expands to fill the available time."

Printer Speed

In the 1960s, when IBM and Xerox were developing the first laser printers that could print an entire page at a time, the goal was to create printers that could print several pages per second so that printer speed could match the performance improvements of computing speed. The computer printouts of that era were all of monospaced font with a small character set of uppercase letters and a few symbols. Although many laser printer designers struggled to produce such simple output with reduced time per page, the product category evolved to produce high quality output for desktop publishing instead of using the improved technology for time reduction. People now wait about as long for a page of printout from a laser printer as they did for a page of printout from the line printers of the 1960s, but the task has been scaled up to full color, high resolution printing encompassing graphics output, and a huge collection of typeset fonts from alphabets in all the world's languages. This is an example of Gustafson's Law applied to printing technology.

Biological Brains

Kevin Howard, of Massively Parallel Technologies Inc., once observed that if Amdahl's Law governed the behavior of biological brains, then a human would have about the same intelligence as a starfish. The human brain has about 100 billion neurons operating in parallel, so for us to avoid passing the point of diminishing returns for all that parallelism, the Amdahl serial fraction f would have to be about 10^{-14}. The fallacy of this seeming paradox is in the underlying assumption that a human brain must do the same task a starfish brain does, but must reduce the execution time to nanoseconds. There is no such requirement, and a human brain accomplishes very little in a few nanoseconds no matter how many neurons it uses at once. Gustafson's Law says that on a time-averaged basis, the human brain will accomplish *vastly more complex tasks* than what a starfish can attempt, and thus avoids the absurd conclusion of the fixed-task model.

Perspective

The concept of scaled speedup had a profound enabling effect on parallel computing, since it showed that simply asking a different question (and perhaps a more realistic one) renders the pessimistic predictions of Amdahl's Law moot. Gustafson's 1988 announcement of 1,000-fold parallel speedup created a turning point in the attitude of computer manufacturers towards massively parallel computing, and now all major vendors provide platforms based on the approach. Most (if not all) of the computer systems in the TOP500 list of the worlds' fastest supercomputers are comprised of many thousands of processors, a degree of parallelism that computer builders regarded as sheer folly prior to the introduction of scaled speedup in 1988.

A common assertion countering Gustafson's Law is that "Amdahl's Law still holds for scaled speedup; it's just that the serial fraction is a lot smaller than had been previously thought." However, this requires inferring the small serial fraction from the measured speedup. This is an example of circular reasoning since it involves choosing a conclusion, then working backward to determine the data that make the conclusion valid. Gustafson's Law is a simple formula that predicts scaled performance from experimentally measurable properties of a workload.

Some have misinterpreted "scaled speedup" as simply increasing the amount of memory for variables, or increasing the fineness of a grid. It is more general than this. It applies to every way in which a calculation can be improved somehow (accuracy, reliability, robustness, etc.) with the addition of more processing power, and then asks *how much longer the enhanced problem would have taken to run without the extra processing power*.

Horst Simon, in his 2005 keynote talk at the International Conference on Supercomputing, "*Progress in Supercomputing: The Top Three Breakthroughs of the Last 20 Years and the Top Three Challenges for the Next 20 Years*," declared the invention of the Gustafson's scaled speedup model as the number one achievement in high-performance computing since 1985.

Related Entries

►Amdahl's Law
►Distributed-Memory Multiprocessor
►Metrics

Bibliographic Entries and Further Reading

Gustafson's 1988 two-page paper in the *Communications of the ACM* [5] outlines his basic idea of fixed-time performance measurement as an alternative to Amdahl's assumptions. It contains the rhetorical question, "How can this be, in light of Amdahl's Law?" that some misinterpreted as a serious plea for the resolution of a paradox. Readers may find a flurry of responses in *Communications* and elsewhere, as well as attempts to "unify" the two laws.

An objective analysis of Gustafson's Law and its relation to Amdahl's Law can be found in many modern textbooks on parallel computing such as [7], [9], or [10]. In much the way some physicists in the early twentieth century refused to accept the concepts of relativity and quantum mechanics, for reasons more intuition-based than scientific, there are computer scientists who refuse to accept the idea of scaled speedup and Gustafson's Law, and who insist that Amdahl's Law suffices for all situations.

Pat Worley analyzed the extent to which one can usefully scale up scientific simulations by increasing their resolution [13]. In related work, Xian-He Sun and Lionel Ni built a more complete mathematical framework for scaled speedup [12] in which they promote the idea of memory-bounded scaling, even though execution time generally increases beyond human patience when the memory used by a problem scales as much as linearly with the number of processors. In a related vein, Vipin Kumar proposed "Isoefficiency" for which the memory increases as much as necessary to keep the efficiency of the processors at a constant level even when communication and other impediments to parallelism are taken into account.

Bibliography

1. Amdahl GM (1967) Validity of the single-processor approach to achieve large scale computing capabilities. AFIPS Joint Spring Conference Proceedings 30 (Atlantic City, NJ, Apr. 18–20), pp 483–485. AFIPS Press, Reston VA. At http://www-inst.eecs.berkeley.edu/~n 252/paper/Amdahl.pdf
2. Bell G (interviewed) (1987) An interview with Gordon Bell. IEEE Software, vol 4, No. 4 (July 1987), pp 102–104
3. Gustafson JL (1990) Fixed time, tiered memory, and superlinear speedup. Distributed Memory Computing Conference, 1990, Proceedings of the Fifth, vol 2 (April 1990), pp 1255–1260. ISBN: 0-8186-2113-3

4. Gustafson JL, Montry GR, Benner RE (1988) Development of parallel methods for a 1024-processor hypercube. SIAM Journal on Scientific and Statistical Computing, vol 9, No. 4, (July 1988), pp 609–638
5. Gustafson (1988) Reevaluating Amdahl's Law. Communications of the ACM, vol 31, No. 5 (May 1988), pp 532–533. DOI=10.1145/42411.42415
6. Heath M, Worley P (1989) Once again, Amdahl's Law. Communications of the ACM, vol 32, No. 2 (February 1989), pp 258–264
7. Hwang K, Briggs F, Computer Architecture and Parallel Processing, 1990. McGraw-Hill Inc., 1990. ISBN: 0070315566
8. Karp A (1985) http://www.netlib.org/benchmark/karp-challenge
9. Lewis TG, El-Rewini H (1992) Introduction to Parallel Computing, Prentice Hall. ISBN: 0-13-498924-4. 32–33
10. Quinn M (1994) Parallel Computing: Theory and Practice. Second edition. McGraw-Hill, Inc
11. Seitz CL (1986) Experiments with VLSI ensemble machines. Journal of VLSI and Computer Systems, vol 1, No. 3, pp 311–334
12. Sun X-H, Ni L (1993) Scalable problems and memory-bounded speedup." Journal of Parallel and Distributed Computing, vol 19, No. 1, pp 22–37
13. Worley PH (1989) The effect of time constraints on scaled speedup. Report ORNL/TM 11031, Oak Ridge National Laboratory

Gustafson–Barsis Law

►Gustafson's Law