داده‌ساختارهای توزیع‌شده‌ی حافظه‌ی خارجی

محمّد قدسی   فرشاد رستم‌آبادی

دانشکده‌ی مهندسی کامپیوتر، دانشگاه صنعتی شریف

۳۰ خرداد ۱۳۸۰

چکیده

حجم بسیار زیاد اطلاعات که امروزه در محاسبات به‌کار می‌رود، نگرش جدیدی را در طراحی الگوریتم‌های کارا به‌وجود آورده است. روشِ سنتیِ طراحیِ الگوریتم عموماً مبتنی بر این است که تمام اطلاعات در حافظه‌ی اصلی جای گیرند. این کار برای حجم بسیار زیاد داده امکان‌پذیر نیست و در چنین صورتی الگوریتم‌های دیگری مورد نیاز است که به آن‌ها «الگوریتم‌های حافظه‌ی خارجی» گفته می‌شود. معیارهای کارایی الگوریتم‌های خارجی با الگوریتم‌های معمول متفاوت است. راه دیگر بالابردن سرعت پردازش، استفاده از پردازش موازی و توزیع‌شده است که در آن اطلاعات حجیم مورد پردازش بین چند کامپیوتر متصل به‌هم توزیع می‌شود و از توان پردازشی موازی می‌توان کارایی بالاتری را به‌دست آورد. محاسباتی که در آن از توزیع‌شدگی و حافظه‌ی خارجی استفاده شده باشد به داده‌ساختارهایی نیاز دارد که به آن «داده‌ساختارهای توزیع‌شده‌ی خارجی» می‌گوییم. در این مقاله این مفاهیم توضیح داده می‌شوند و الگوریتم‌هایی توزیع‌شده و خارجی برای چند مسئله ارائه و تحلیل می‌شوند.

۱ معرفی

الگوریتم‌های حافظه‌ی خارجی به آن دسته از الگوریتم‌ها گفته می‌شود که به‌علت حجم بسیار زیاد داده‌ی ورودی، انتقال یک‌جای آن به حافظه‌ی اصلی امکان‌پذیر نیست و این کار باید در چند مرحله و در قطعات کوچک‌تر صورت گیرد. به‌علت آن‌که انتقال داده از حافظه‌ی خارجی به اصلی بسیار کند است، الگوریتم‌های کارای خارجی باید کم‌ترین تعداد دسترسی به حافظه‌ی خارجی را داشته باشد.

راه دیگر برای بالا بردن کارایی حل مسئله‌هایی با حجم زیاد داده، استفاده از توزیع‌شدگی و پردازش موازی است. با توزیع داده‌ها بر روی چند پردازنده و استفاده هم‌زمان از دیسک‌های آن‌ها و با ترکیب الگوریتم‌های توزیع شده و حافظه‌ی خارجی، می‌توان به زمانِ پردازش کوتاه‌تری دست پیدا کرد. برای رسیدن به این هدف، علاوه بر طراحی یک

الگوریتم خارجی و توزیع شده به داده‌ساختارهای توزیع شده[1] نیاز است تا بتوان مشکلات توزیع‌شدگی را از دید طراح الگوریتم پنهان ساخت.

در این مقاله ابتدا مدل محاسباتی حافظه‌ی خارجی تشریح می‌شود. سپس یک الگوریتم مرتب‌ساز حافظه‌ی خارجی بر اساس الگوریتم ادغام[2] تشریح می‌شود که ایده‌های اصلی آن بعداً مورد استفاده قرار می‌گیرد. بخش ۴، به معرفی یک داده‌ساختار[3] حافظه‌ی خارجی به‌نام درخت میان‌گیر[4] می‌پردازد. در بخش‌های ۵، ٦ و ۷ چند مسئله‌ی پایه‌ای حافظه‌ی خارجی و داده‌ساختارهای مورد نیاز برای حل آن‌ها ارائه می‌شود. این مسئله‌ها پایه‌ای برای الگوریتم توزیع‌شده‌ی حافظه‌ی خارجی جدیدی برای مسئله‌ی پیدا کردن تلاقی خطوط هستند که حالت خاص آن با خطوط افقی و عمودی در بخش ۷ ارائه می‌شود.

۲ مدل محاسباتی حافظه خارجی

مدل دیسک‌های موازی که توسط [۷] ارائه شده است، یکی از مدل‌های رایج در پردازش داده‌های خارجی است. در این مدل، فرض شده است که یک پردازنده دارای تعدادی دیسک سخت به‌عنوان حافظه‌ی خارجی می‌باشد که می‌تواند به‌صورت هم‌زمان به آن‌ها دسترسی داشته باشد. به‌علاوه پارامترهای زیر نیز در این مدل لحاظ شده‌اند:

N : تعداد داده‌ی مسئله
M : تعداد داده‌ای که در حافظه‌ی اصلی جای می‌گیرند
B : تعداد داده‌ی یک قطعه از دیسک
D : تعداد دیسک‌های سخت

به‌طوری‌که $M < N$ و $1 < DB < M/2$. پارامترهای دیگر عبارتند از: $m = M/DB$ و $n = N/DB$.

۳ مرتب‌سازیِ خارجی بر اساس ادغام

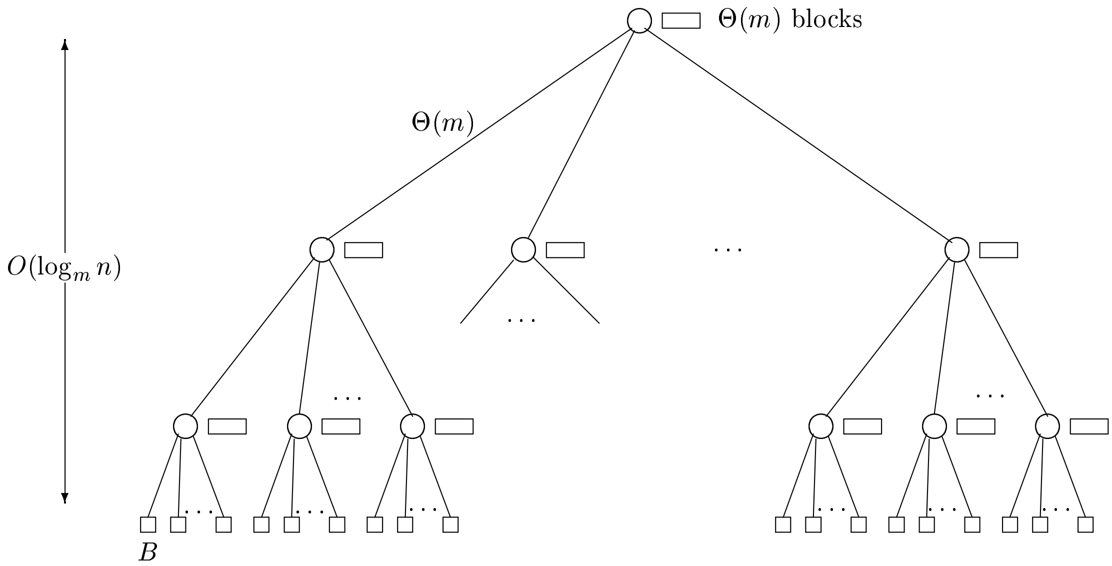
این الگوریتم مرتب سازی بر اساس Merge Sort بنا شده است [۳]. در این روش، فرض می‌شود در ابتدا داده‌ها در یک بسته به‌اندازه‌ی واحد قرار دارند که هر بسته مرتب است. سپس داده‌ها به دو بخش مساوی تقسیم شده و مراحل زیر تکرار می‌شود:

۱) دو بخش خالی برای افزودن قطعات بعدی ایجاد کن.

۲) داده‌های اولین قطعه هر یک از دو بخش را به‌ترتیب خوانده و در یکدیگر ادغام کن و یک قطعه‌ی جدید تولید کن. قطعات جدید را یک در میان در بخش های تولید شده در مرحله ۱ بنویس.

۳) مرحله ۲ را تکرار کن تا تمام قطعات دو بخش ورودی در یکدیگر ادغام شوند. سپس دو بخش ورودی را حذف و بخش‌های پر شده در مرحله ۲ را به‌عنوان ورودی برای تکرار بعدی الگوریتم استفاده کن. اگر یکی از دو بخش خالی است، عمل مرتب‌سازی خاتمه می‌یابد.

[1] distributed data structures
[2] merge
[3] data structures
[4] buffer tree



زمان اجرای این الگوریتم از مرتبه $O\big((N\log N)/B\big)$ است زیرا در هر تکرار، تمامی داده‌ها یک‌بار مورد پردازش قرار می‌گیرند و تعداد تکرار الگوریتم نیز برابر $O\big(\log N\big)$ است. اگر به‌جای شروع از قطعات با اندازه‌ی واحد، از قطعات با اندازه‌ی M استفاده می‌شد، تعداد تکرار الگوریتم تا $O\big(\log(N/B)\big)$ کاهش می‌یابد و زمان اجرای نهایی از مرتبه $O\big((N/B)\log(N/B)\big)$ می‌شود.

از معایب این روش باید به عدم امکان درج (حذف) یک عنصر به (از) یک فهرست مرتب از داده‌ها بدون نیاز به مرتب‌سازی مجدد تمامی داده‌ها اشاره کرد.

۴ داده‌ساختار درخت میانگیر

این داده‌ساختار بر اساس درخت «بی» طراحی شده است [۱]. در این داده‌ساختار درج (حذف) یک عنصر در (از) مکان صحیح خود می‌تواند در زمان $O(\log_m n)$ انجام شود. زمان کل اجرای یک الگوریتم مرتب‌سازی با استفاده از این داده‌ساختار از مرتبه‌ی $O(n\log_m n)$ است.

در داده‌ساختار درخت «بی»، به‌ازای هر درج یا حذف، نیاز به یک‌بار حرکت از ریشه به برگ متناظر و بر عکس وجود دارد. به‌عبارت دیگر برای درج N عنصر به یک درخت «بی»، $O(N\log n)$ دسترسی به دیسک مورد نیاز است. برای کاهش تعداد دسترسی به دیسک تا حد مطلوبِ $O(n\log_m n)$، به‌ازای هر گره‌ی درخت «بی»، یک حافظه میانگیر[5] به‌اندازه‌ی $M/2$ روی دیسک ایجاد می‌کنیم و از یک نوع روش تنبل[6] برای درج عناصر استفاده می‌شود.

داده‌ساختار درخت میانگیر یک درخت «بی» است که هر گره‌ی آن حداکثر دارای $m/2$ فرزند است. برای درج یک عنصر به یک گره، ابتدا عنصر به حافظه‌ی میانگیر متصل به گره اضافه می‌شود که این کار به حداکثر $O(1)$ دسترسی به دیسک نیاز دارد. در صورت پر شدن حافظه‌ی میانگیر، از الگوریتم تخلیه‌ی حافظه‌ی میانگیر استفاده می‌شود. برای تخلیه‌ی‌یک حافظه‌ی میانگیر، ابتدا تمام حافظه‌ی میانگیر به‌همراه گره‌ی مرتبط آن به حافظه‌ی اصلی آورده می‌شود. کل حافظه‌ی مورد نیاز برای این کار $M/2$ برای میانگیر و $m/2$ برای گره است. سپس داده‌های حافظه‌ی میانگیر در

[5] buffer
[6] lazy

داخل حافظه‌ی اصلی مرتب شده و با استفاده از کلیدهای درون گره، داده‌هایی که برای درج باید به هر یک از بچه‌های این گره ارسال شوند تعیین می‌شود و به‌صورت بازگشتی به آن‌ها افزوده می‌شود.

چون یک بار تخلیه‌ی یک حافظه‌ی میان‌گیر به یک سطح پایین‌تر از درخت به زمان $O(m)$ نیاز دارد و نیز چون به‌طور متوسط هر بلاک از داده به‌تعداد ثابتی در هر یک از $O(\log_m n)$ سطح درخت دیده می‌شود، لذا می‌توان نتیجه گرفت که تمام n داده‌ی مورد نظر را می‌توان با $O(n \log_m n)$ دسترسی به دیسک، درج نمود.

۱.۴ داده‌ساختار صف اولویت در حافظه‌ی خارجی

برای پیاده‌سازی یک صف الویت[۷] در حافظه‌ی خارجی می‌توان از داده‌ساختار متداول Heap استفاده کرد. با استفاده از این مدل و پیاده‌سازی آرایه‌ای آن، می‌توان n عمل درج و حذف را با $O(n \log n)$ دسترسی به دیسک و k عمل «حذف کوچک‌ترین» را با $O\big(k \log n/B\big)$ دسترسی به دیسک انجام داد.

با استفاده از درخت میانگیر برای یک عمل حذف کوچک‌ترین به $O(\log_m n)$ دسترسی به دیسک، پس از ساخت درخت، نیاز است. ولی با توجه به این‌که تمام B داده‌ی درونِ سمت چپ‌ترین بلاکِ کوچک‌ترین عناصر کل داده‌ها است، می‌توان با نگهداری این بلاک در حافظه‌ی اصلی می‌توان بدون دسترسی به دیسک $B - ۱$ عمل «حذف کوچک‌ترین» بعدی را جواب داد. بنابراین به‌طور متوسط برای هر عمل «حذف کوچک‌ترین» به $O\big(\log_m n/B\big)$ دسترسی به دیسک نیاز است.

۵ مسئله‌ی تسلط بر نقطه (EPD)

مسئله‌ی تسلط بر نقطه[۸] به‌صورت زیر تعریف می‌شود:

> تعدادی خط راست غیر متقاطع که هر خط با زوج نقاط شروع و خاتمه‌ی آن مشخص می‌شود داده شده‌اند. می‌خواهیم به‌ازای هر نقطه‌ی انتهایی هر خط، پایین‌ترین خطی را که دقیقاً بالای آن نقطه قرار دارد پیدا کنیم.

الگوریتم‌های حل این مسئله در حافظه‌ی اصلی به‌صورت کارآ در زمان متناسب با $O(N \log N)$ جواب را پیدا می‌کنند. روش‌هایی نیز برای حل این مسئله در حافظه‌ی خارجی ارائه شده‌اند که از تکنیک‌های متفاوتی از جمله «جاروی توزیعی[۹]» و درخت میانگیر استفاده کرده‌اند.این روش‌ها به‌تنهایی برای حل این مسئله کافی نیستند.

برای حل کامل این مسئله باید از ترکیب روش‌های فوق استفاده کرد. نشان می‌دهیم که این روشِ ترکیبی با تعدادِ عملیاتِ دیسکِ متناسب با $O(n \log_m n)$ می‌تواند به جواب برسد.

مسئله‌ی EPD کاربرد فراوانی دارد که از جمله به مسائل زیر اشاره می‌شود:

- تجزیه‌ی ذوزنقه‌وار N خط غیر متقاطع با $O(n \log_m n)$ عملیاتِ دیسک
- مثلث‌بندی یک چندضلعی ساده با $O(n \log_m n)$ عملیاتِ دیسک
- پیداکردن ترتیبِ کلی برای N خط غیر متقاطع با $O(n \log_m n)$ عملیاتِ دیسک

[۷] priority queue
[۸] Endpoint Dominance Problem
[۹] distribution sweeping

۱.۵ روش درخت میانگیر و جاروی توزیعی

در حافظه‌ی اصلی می‌توان با یک الگوریتم ساده به جواب رسید: صفحه را با یک خط عمودی از چپ به راست جارو می‌کنیم. با رسیدن به نقطه‌ی ابتدای هر خط، آن را در لیست مرتب خطوط فعال وارد و با رسیدن به نقطه‌ی انتهایی خط، آن را از لیست خارج می‌کنیم. به‌علاوه، با دیدن هر نقطه به‌کمک یک جستجو در لیست خطوط می‌توان خطی را که دقیقاً بالای آن قرار دارد، تعیین کرد[۹].

درخت میان‌گیر می‌تواند N عمل به‌هنگام سازی[۱۰] و K عمل جستجو را روی یک درخت که در ابتدا خالی است با $O((n+k)\log_m n)$ عملیات دیسک انجام دهد. این درخت می‌تواند در الگوریتمی که برای حافظه‌ی اصلی طراحی شده بود، قرار گرفته و یک الگوریتم حافظه‌ی خارجی ایجاد نماید[۴]. اشکال اصلی این روش در نگهداری مرتب خطوط می‌باشد، چراکه ترتیبِ کاملاً مشخصی برای خطوط غیر متقاطع در دست نداریم.

یک روش پر استفاده‌ی در حافظه‌ی خارجی، جاروی توزیعی است [۴]. در حالت کلی این روش ابتدا فضای مورد نظر را به m باریکه‌ی عمودی تقسیم می‌کند به طوری‌که در هر یک $\Theta(n/m)$ داده قرار گیرند. سپس به‌صورت بازگشتی جواب هر باریکه را محاسبه و سپس جواب‌ها در یک‌دیگر ادغام می‌نماید. برای استفاده از این روش می‌باید یک بار خطوط را مرتب نمود تا در تکرارهای بعدی الگوریتم نیازی به مرتب سازی مجدد آن‌ها نباشد که این همان مشکل استفاده از روش درخت میانگیر است.

۶ درخت پاره‌خط‌ها در حافظه‌ی خارجی

درخت پاره‌خط‌ها[۱۱] یک داده‌ساختارِ شناخته شده برای نگهداری تعدادی پاره‌خط در یک بُعد می‌باشد به‌طوری‌که به‌ازای هر نقطه‌ی دلخواه می‌توان فهرست تمام پاره‌خط‌هایی که شامل آن نقطه می‌باشند را استخراج نمود. نسخه‌ی حافظه‌ی خارجی این درخت را می‌توان با استفاده از درخت میانگیر تولید نمود [۵، ۴].

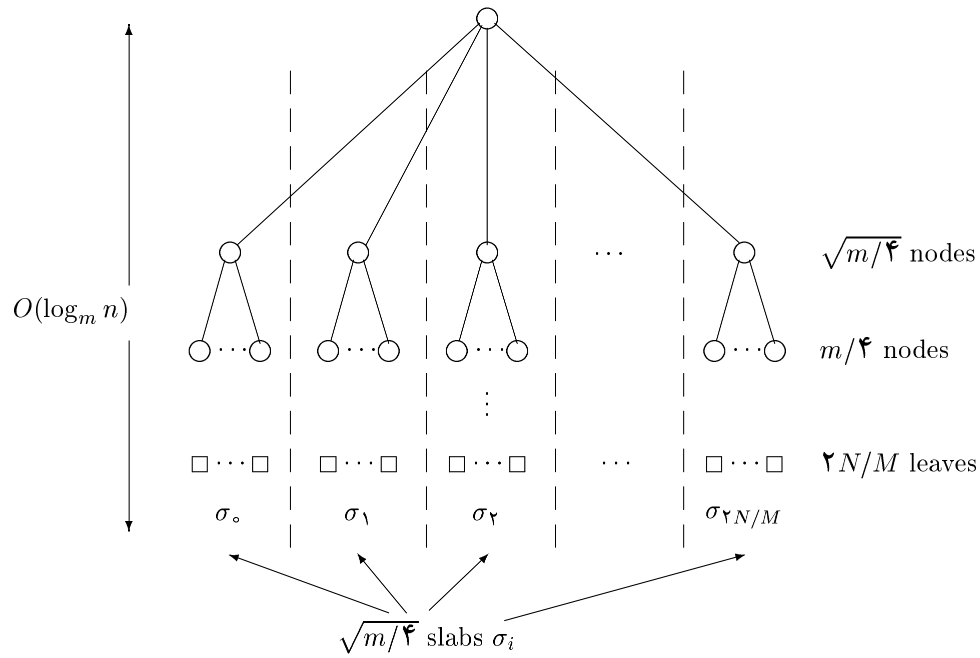
برای ساخت درخت پاره‌خط‌ها در حافظه‌ی خارجی، ابتدا کل بازه به $\sqrt{m/۴}$ قطعه تقسیم می‌شود و در نتیجه اولین سطح درخت دارای $\sqrt{m/۴}$ بازه‌ی σ_i می‌باشد. بازه‌ی ترکیبی $[\sigma_i, \sigma_j], i < j$ ترکیب بازه‌های σ_i الی σ_j می‌باشد.

پاره‌خط «بلند» پاره‌خطی است که در بیش از دو بازه قرار گیرد. این پاره‌خط در بزرگ‌ترین بازه‌ی ترکیبی که کاملاً آن را می‌پوشاند ذخیره می‌شود. دو بخش انتهایی یک پاره‌خط بلند، در بازه‌های ابتدایی و انتهایی دربرگیرنده خط به‌صورت دو پاره‌خط مجازی جدید تعریف می‌شوند. این پاره‌خط‌های مجازی در تکرارهای بازگشتی بعدی به پاره‌خط‌های بلند تبدیل و در گره‌ی متناظر خود ذخیره می‌شوند. پاره‌خط «کوتاه» پاره‌خطی است که حداکثر در دو بازه قرار گیرد. پاره‌خط‌های کوتاه نیز در تکرارهای بازگشتی بعدی درخت ذخیره می‌شوند.

۱.۶ ساختن درخت پاره‌خط‌ها در حافظه خارجی

یک روش ساده برای ساخت درخت پاره‌خط‌ها به صورت زیر است: ابتدا تمام نقاط را بر حسب x مرتب کنید. این فهرست، در تمام تکرارهای الگوریتم مورد استفاده قرار می‌گیرد. نقاط را به $\sqrt{m/۴}$ بازه‌ی مساوی تقسیم کن. سپس برای ساخت $O(m)$ بازه‌ی ترکیبی در ریشه، تمام خطوط را بررسی کن و هر خط بلند را در فهرستِ خطوطِ بازه‌ی ترکیبی متناظر خود قرار ده. سپس تمام فهرست‌های بازه‌های ترکیبی را با استفاده از یک الگوریتم مرتب‌سازی بهینه مرتب کن.

[۱۰] update
[۱۱] segment tree



این کار را برای تمام فرزندان به‌صورت بازگشتی انجام ده تا تعداد نقاط یک گره به زیر $M/۲$ برسد. تعداد عملیات دیسک این الگوریتم متناسب با $O(n \log_m^۲ n)$ می‌باشد.

برای کاهش تعداد دسترسی به دیسک این الگوریتم به $O(n \log_m n)$ تغییراتی در نحوه‌ی ساخت این درخت داده شده است. برای ساخت یک گره، علاوه بر داشتن فهرست مرتب پاره‌خط‌های درون آن، فرض می‌شود که دو فهرست مرتبِ دیگر نیز در دسترس است: فهرست مرتب اول شامل پاره‌خط‌هایی است که فقط نقطه‌ی شروع آن‌ها در بازه‌ی گره‌ی جاری قرار دارد و فهرستِ مرتبِ دوم شامل پاره‌خط‌هایی است که فقط نقطه‌ی خاتمه‌ی آن‌ها در بازه‌ی گره جاری قرار دارد.

حال ابتدا با فرض داشتن فهرست‌های مرتب برای یک گره، فهرست‌های مرتب بازه‌های ترکیبی آن گره را تولید می‌کنیم، سپس فهرست‌های مرتب مورد نیاز فرزندان همان گره را می‌سازیم تا الگوریتم بتواند به‌صورت بازگشتی به ساختن درخت ادامه دهد.

برای ساخت یک گره، فهرست نامرتب خطوط را بررسی می‌کنیم و هر یک را در بازه‌ی ترکیبی متناظر خود قرار می‌دهیم. سپس فهرست‌های خطوط بازه‌های ترکیبی را با استفاده از یک الگوریتم مرتب‌ساز بهینه، مرتب می‌کنیم. سپس دو فهرست مرتب ورودی را بررسی و خطوط آن‌ها را به فهرست‌های مرتب‌شده‌ی بازه‌های ترکیبی اضافه می‌کنیم. این کار متناسب با یک‌بار خواندنِ فهرست‌ها به عملیات دیسک نیاز دارد. حال که اطلاعات مورد نیاز گره‌ی جاری تأمین شد، به تولید فهرست‌های ورودی فرزندانِ این گره می‌پردازیم. به‌سادگی دیده می‌شود که با داشتن فهرست‌های مرتبِ بازه‌های ترکیبی، می‌توان با صرفِ حافظه‌ای متناسب با $\sqrt{m}$، فهرست‌های مرتب هر فرزند از گره‌ی جاری را تولید نمود. توجه کنید که هر گره‌ی دارای $\sqrt{m/۴}$ فرزند است و باید $۲\sqrt{m/۴}$ فهرست مرتب تولید کند و با توجه به این‌که حافظه‌ی اصلی گنجایش m داده را در خود دارد، لذا حافظه‌ی کافی برای تولید فهرست‌های مرتب وجود دارد.

تعداد عملیات دیسک مورد نیاز برای ساختن درخت پاره‌خط‌ها با روش فوق متناسب با $O(n \log_m n)$ است، چرا که هر پاره خط تنها یک‌بار در فهرست غیر مرتبی قرار می‌گیرد که باید مرتب شود.

۷ پیدا کردن تلاقی دو مجموعه‌ی خطوط

حال با داشتن دو مجموعه از پاره‌خط‌ها که هر کدام غیر متقاطع می‌باشند، می‌خواهیم مجموعه‌ی پاره‌خط‌هایی که از تلاقی این دو مجموعه حاصل می‌شوند را محاسبه کنیم[۱۲].

فرض کنید که دو مجموعه‌ی پاره‌خط‌های ورودی را به صورت S_r و S_b نمایش دهیم. حال مجموعه‌های زیر را تعریف می‌کنیم:

$$T_r = S_r \cup \bigcup_{(p,q)\in S_b} \{(p,p),(q,q)\}$$

$$T_b = S_b \cup \bigcup_{(p,q)\in S_r} \{(p,p),(q,q)\}$$

دیده می‌شود که اندازه‌ی هر دو مجموعه برابر است با $O(|S_r|+|S_b|) = O(N)$. حال هر دو مجموعه را با استفاده از درخت پاره‌خط‌ها با $O(n \log_m n)$ عملیات دیسک مرتب می‌کنیم. سپس با استفاده از روش جاروی توزیعی با $\sqrt{m}$ فرزند برای هر گره، نقاط تلاقی را به‌دست می آوریم.

برای ساخت درخت مورد نیاز جاروی توزیعی صفحه را به $\sqrt{m}$ باریکه تقسیم می‌کنیم و با استفاده از روش توضیح داده شده در ۱.۶ به تولید درخت می‌پردازیم.

البته برای حالت مورد نظر ما یک خط «بلند» را خطی تعریف می‌کنیم که در در بیش از یک باریکه و خط «کوتاه» خطی است که کاملاً در یک باریکه قرار می‌گیرد.

۸ الگوریتم توزیع شده‌ی پیدا کردن نقاط تلاقی دو مجموعه‌ی خطوط نامتقاطع

مسئله‌ی پیدا کردن نقاط تلاقی، که ما آن را در حالت خاص خطوط افقی و عمودی در فضای دو بعدی بررسی می‌کنیم، با استفاده از یک الگوریتم ساده در حافظه‌ی اصلی به صورت کارآ قابل حل است [۹]. حالت حافظه‌ی خارجی مسئله نیز در [۸] بررسی شده است. ما این مسئله را برای حالتی که یک محیط توزیع شده در دسترس می‌باشد حل می‌کنیم.

۱.۸ الگوریتم توزیع شده

این الگوریتم بر اساس الگوریتم [۲] طراحی شده است.

در این الگوریتم یک درخت توزیع‌شده ساخته می‌شود که در ریشه‌ی خود p فرزند (در پردازنده‌ی اصلی) دارد و فرزند i آن (متناظر با پردازنده‌ی کمکی i)، که دارای حافظه‌ی اصلی متناسب با $O(m_i)$ است، دارای $O(\sqrt{m}_i)$ فرزند می‌باشد.

۲.۸ پردازنده‌های مورد نیاز

در این الگوریتم به p پردازنده‌ی کمکی نیاز است که توسط یک پردازنده‌ی اصلی هماهنگ می‌شوند. پردازنده‌ی اصلی تنها برای خواندن داده‌های ورودی و ثبت خروجی به دیسک نیاز دارد و در زمان اجرای برنامه به حافظه‌ی خارجی

[۱۲] Line Intersection

مستقلی احتیاج ندارد، به‌علاوه با توجه به کم بودن وظایف این پردازنده، می‌توان کارهای آن را در یک پردازنده‌ی دیگر ادغام نموده و تنها با استفاده از p پردازنده الگوریتم را اجرا نمود.

۳.۸ الگوریتم پردازنده‌ی اصلی

پردازنده‌ی اصلی ابتدا دوبار الگوریتم مرتب‌سازی را بر روی نقاط انتهایی خطوط انجام می‌دهد. یک‌بار آن‌ها را بر حسب x و بار دیگر بر حسب y به‌صورت توزیع‌شده مرتب می‌کند. سپس با استفاده از فهرست مرتب‌شده‌ی اول، مرزهای p باریکه‌ی عمودی که تعداد مساوی نقاط در آن قرار می‌گیرند را تعیین می‌کند و به هر یک از p پردازنده، یک باریکه تخصیص می‌دهد و بخش مرتب‌شده از فهرست x متناظر با هر پردازنده را برای آن ارسال می‌کند.

به‌سادگی دیده می‌شود که نیازی به ذخیره کردن فهرست مرتب x در پردازنده‌ی اصلی وجود ندارد. به‌علاوه فهرست مرتب y ها نیز در زمان استفاده تولید و مصرف می‌شود و نیازی به نگهداری آن نیست.

پردازنده‌ی اصلی در گام بعد با استفاده از فهرست y ها، یک جاروی عمودی بالا به پایین را رهبری می‌کند. برای این کار به‌ترتیب از این فهرست، خطوط خوانده می‌شوند. برای هر خط خوانده‌شده، و برای پردازنده‌هایی که دارای بازه‌ی x این خط می‌باشند، ارسال می‌گردد. البته در پردازنده‌ی اصلی اگر خطی در بیش از یک باریکه قرار گیرد به تمام پردازنده‌هایی که خط در آن‌ها قرار گرفته است ارسال می‌گردد (بر خلاف حالت مشابه در پردازنده‌های کمکی). در انتها نتایج جمع شده در پردازنده‌های کمکی، توسط پردازنده‌ی اصلی جمع‌آوری می‌شود.

۴.۸ الگوریتم پردازنده‌های کمکی

پردازنده‌های کمکی ابتدا فهرست مرتب شده x ناحیه‌ی خود را دریافت می‌کنند تا در مراحل ساخت درخت خود بتوانند از آن استفاده کنند.

در مرحله‌ی بعد هر پردازنده‌ی کمکی دقیقاً الگوریتم [۲] را انجام می‌دهد، با این تفاوت که در هر مرحله از الگوریتم داده‌ها از طریق پردازنده‌ی اصلی دریافت می‌شوند. در انتها نتایج برای پردازنده‌ی اصلی ارسال می‌گردد.

۵.۸ مرتب‌سازی توزیع شده

برای مرتب‌سازی توزیع شده، از یک الگوریتم ساده استفاده می‌شود. ابتدا پردازنده اصلی داده‌های خود را بین p پردازنده تقسیم می‌کند. سپس هر پردازنده با استفاده از یک الگوریتم مرتب‌سازی حافظه‌ی خارجی، داده‌های خود را مرتب می‌نماید.

پس از مرتب شدن داده‌ها در هر پردازنده، پردازنده‌ها داده‌های خود را به پردازنده اصلی ارسال می‌کنند. پردازنده‌ی اصلی نیز با استفاده از یک الگوریتم ادغام فهرست مرتب اصلی را تولید می‌نماید.

۶.۸ زمان اجرای الگوریتم

زمان اجرای کلی الگوریتم برابر با حاصل‌جمعِ زمانِ مرتب‌سازی‌ها، ارسال خطوط و پردازش آن‌ها توسط پردازنده‌های کمکی و جمع‌آوری نتایج می‌باشد. زمان مورد نیاز برای مرتب‌سازی برای n داده با استفاده از p پردازنده‌ی کمکی برابر با $O(n/p\log_m(n/p)+n)$ می‌باشد. زمان مورد نیاز برای پردازش خطوط ارسالی نیز متناسب با $O(n/p\log_m(n/p)+t)$ می‌باشد که t تعداد نقاط تلاقی پیدا شده است [۲]. زمان‌های ارسال و دریافت داده‌ها همگی از $O(n)$ می‌باشند که در

مجموع زمان اجرای $O(n/p \log_m(n/p) + t + n)$ را به‌دست می‌دهد. با اندکی دقت دیده می‌شود که فضای مورد نیاز این الگوریتم نیز از مرتبه‌ی $O(n+t)$ می‌باشد. البته در پردازنده‌های کمکی به فضایی متناسب با $O(n/p)$ نیاز است.

۷.۸ موازنه‌ی بار پردازنده‌ها

با فرض برابر بودن سرعت پردازنده‌ها، باید بار هر پردازنده متناسب با حافظه‌ی اصلی در دسترس آن تعیین گردد. با توجه به درخت توزیع‌شده‌ای که توسط الگوریتم ساخته می‌شود، کافی است که هر پردازنده با توجه به مقدار حافظه‌ی اصلی خود تعداد گره‌های هر سطح درخت خود را تعیین نماید.

اگر پردازنده‌ی i دارای حافظه‌ی اصلی متناسب با $O(m_i)$ است آن‌گاه حداکثر تعداد بچه‌های هر گره را متناسب با $O(\sqrt{m_i})$ قرار می‌دهیم. استفاده از $O(\sqrt{m_i})$ به‌جای $O(m_i)$، ارتفاع درخت ساخته شده توسط این پردازنده را دو برابر می‌کند که تأثیری بر زمان اجرای الگوریتم نخواهد داشت.

۹ خلاصه و نتیجه‌گیری

در این مقاله چند داده‌ساختار و الگوریتم حافظه‌ی خارجی برای افزودن امکانات توزیع‌شدگی بررسی شد و در پایان الگوریتم پیدا کردن تلاقی خطوط به همراه داده‌ساختار توزیع شده‌ی حافظه‌ی خارجی مورد نیاز آن طراحی و تحلیل شد. این الگوریتم در آزمایشگاه پردازش موازی دانشکده‌ی کامپیوتر پیاده‌سازی شد.

در ادامه‌ی این طرح می‌توان با بررسی سایر الگوریتم‌های حافظه‌ی خارجی و تحلیل داده‌ساختارهای موردنیاز آن‌ها، الگوریتم و داده‌ساختارهای کارآی بیشتری برای حل این‌گونه مسائل تهیه نمود.

مراجع

1. L. Arge, The buffer tree: A new technique for optimal I/O-algorithms, *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334-345, 1995.

2. L. Arge, Efficient External-Memory Data Structures and Application, *PhD Thesis*, Aarhus, Feburary/Augest 1996.

3. L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, On sorting strings on external memory, *Proc. ACM Symp, on Theory of computation*, pp. 540-548, 1997.

4. L. Arge, D. E. Vengroff, and J. S. Vitter, External memory algorithms for processing line segments in geographical information system, In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pp. 295-310, 1995.

5. J. L. Bentley, and T. A. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Transactions on Computers*, C-28(9):643-647, 1979.

6. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tammisa, D. E. Vengrott, and J. S. Vitter, External-Memory graph alghorithms, *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 134-149, 1995.

7. T. H. Corman, C. E. Leiserson, and R. L. Rivest Introduction to Algorithms, *MIT Press*, 1990.

8. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter External memory computational geomery, *Proc. IEEE Symp, on Foundation of Comp. Sci.*, pp. 714-723, 1993.

9. F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, *Springer-Verlog*, 1985.